Contents

SQL Server and ADO.NET

4/26/2022 • 2 minutes to read • Edit Online

This section describes features and behaviors that are specific to the .NET Framework Data Provider for SQL Server (System.Data.SqlClient).

System.Data.SqlClient provides access to versions of SQL Server, which encapsulates database-specific protocols. The functionality of the data provider is designed to be similar to that of the .NET Framework data providers for OLE DB, ODBC, and Oracle. System.Data.SqlClient includes a tabular data stream (TDS) parser to communicate directly with SQL Server.

> **NOTE**
>
> To use the .NET Framework Data Provider for SQL Server, an application must reference the System.Data.SqlClient namespace.

In This Section

SQL Server Data Types and ADO.NET
Describes how to work with SQL Server data types and how they interact with .NET Framework data types.

SQL Server Binary and Large-Value Data
Describes how to work with large value data in SQL Server.

SQL Server Data Operations in ADO.NET
Describes how to work with data in SQL Server. Contains sections about bulk copy operations, MARS, asynchronous operations, and table-valued parameters.

SQL Server Features and ADO.NET
Describes SQL Server features that are useful for ADO.NET application developers.

LINQ to SQL
Describes the basic building blocks, processes, and techniques required for creating LINQ to SQL applications.

For complete documentation of the SQL Server Database Engine, see SQL Server technical documentation.

See also

- Securing ADO.NET Applications
- Data Type Mappings in ADO.NET
- DataSets, DataTables, and DataViews
- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

SQL Server Data Types and ADO.NET

4/26/2022 • 2 minutes to read • Edit Online

SQL Server and the .NET Framework are based on different type systems, which can result in potential data loss. To preserve data integrity, the .NET Framework Data Provider for SQL Server (System.Data.SqlClient) provides typed accessor methods for working with SQL Server data. You can use the enumerations in the SqlDbType classes to specify SqlParameter data types.

For more information and a table that describes the data type mappings between SQL Server and .NET Framework data types, see SQL Server Data Type Mappings.

SQL Server 2008 introduces new data types that are designed to meet business needs to work with date and time, structured, semi-structured, and unstructured data. These are documented in SQL Server 2008 Books Online.

The SQL Server data types that are available for use in your application depends on the version of SQL Server that you are using. For more information, see Data Types (Transact-SQL).

In This Section

SqlTypes and the DataSet
Describes type support for `SqlTypes` in the `DataSet`.

Handling Null Values
Demonstrates how to work with null values and three-valued logic.

Comparing GUID and uniqueidentifier Values
Demonstrates how to work with GUID and uniqueidentifier values in SQL Server and the .NET Framework.

Date and Time Data
Describes how to use the new date and time data types introduced in SQL Server 2008.

Large UDTs
Demonstrates how to retrieve data from large value UDTs introduced in SQL Server 2008.

XML Data in SQL Server
Describes how to work with XML data retrieved from SQL Server.

Reference

DataSet
Describes the `DataSet` class and all of its members.

System.Data.SqlTypes
Describes the `SqlTypes` namespace and all of its members.

SqlDbType
Describes the `SqlDbType` enumeration and all of its members.

DbType
Describes the `DbType` enumeration and all of its members.

See also

- SQL Server Data Type Mappings
- Configuring Parameters and Parameter Data Types
- Table-Valued Parameters
- SQL Server Binary and Large-Value Data
- ADO.NET Overview

SqlTypes and the DataSet

4/26/2022 • 3 minutes to read • Edit Online

ADO.NET 2.0 introduced enhanced type support for the `DataSet` through the System.Data.SqlTypes namespace. The types in System.Data.SqlTypes are designed to provide data types with the same semantics and precision as the data types in a SQL Server database. Each data type in System.Data.SqlTypes has an equivalent data type in SQL Server, with the same underlying data representation.

Using System.Data.SqlTypes directly in a DataSet confers several benefits when working with SQL Server data types. System.Data.SqlTypes supports the same semantics as SQL Server native data types. Specifying one of the System.Data.SqlTypes in the definition of a DataColumn eliminates the loss of precision that can occur when converting decimal or numeric data types to one of the common language runtime (CLR) data types.

Example

The following example creates a DataTable object, explicitly defining the DataColumn data types by using System.Data.SqlTypes instead of CLR types. The code fills the DataTable with data from the Sales.SalesOrderDetail table in the AdventureWorks database in SQL Server. The output displayed in the console window shows the data type of each column, and the values retrieved from SQL Server.

```
static private void GetSqlTypesAW(string connectionString)
{
    // Create a DataTable and specify a SqlType
    // for each column.
    DataTable table = new DataTable();
    DataColumn icolumnolumn =
        table.Columns.Add("SalesOrderID", typeof(SqlInt32));
    DataColumn priceColumn =
        table.Columns.Add("UnitPrice", typeof(SqlMoney));
    DataColumn totalColumn =
        table.Columns.Add("LineTotal", typeof(SqlDecimal));
    DataColumn columnModifiedDate =
        table.Columns.Add("ModifiedDate", typeof(SqlDateTime));

    // Open a connection to SQL Server and fill the DataTable
    // with data from the Sales.SalesOrderDetail table
    // in the AdventureWorks sample database.
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        string queryString =
            "SELECT TOP 5 SalesOrderID, UnitPrice, LineTotal, ModifiedDate "
            + "FROM Sales.SalesOrderDetail WHERE LineTotal < @LineTotal";

        // Create the SqlCommand.
        SqlCommand command = new SqlCommand(queryString, connection);

        // Create the SqlParameter and assign a value.
        SqlParameter parameter =
            new SqlParameter("@LineTotal", SqlDbType.Decimal);
        parameter.Value = 1.5;
        command.Parameters.Add(parameter);

        // Open the connection and load the data.
        connection.Open();
        SqlDataReader reader =
            command.ExecuteReader(CommandBehavior.CloseConnection);
        table.Load(reader);

        // Close the SqlDataReader.
        reader.Close();
    }

    // Display the SqlType of each column.
    Console.WriteLine("Data Types:");
    foreach (DataColumn column in table.Columns)
    {
        Console.WriteLine(" {0} -- {1}",
            column.ColumnName, column.DataType.UnderlyingSystemType);
    }

    // Display the value for each row.
    Console.WriteLine("Values:");
    foreach (DataRow row in table.Rows)
    {
        Console.Write(" {0}, ", row["SalesOrderID"]);
        Console.Write(" {0}, ", row["UnitPrice"]);
        Console.Write(" {0}, ", row["LineTotal"]);
        Console.Write(" {0} ", row["ModifiedDate"]);
        Console.WriteLine();
    }
}
```

```
Private Sub GetSqlTypesAW(ByVal connectionString As String)

    ' Create a DataTable and specify the
    ' SqlType for each column.
    Dim table As New DataTable()
    Dim icolumnolumn As DataColumn = _
      table.Columns.Add("SalesOrderID", GetType(SqlInt32))
    Dim priceColumn As DataColumn = _
      table.Columns.Add("UnitPrice", GetType(SqlMoney))
    Dim totalColumn As DataColumn = _
      table.Columns.Add("LineTotal", GetType(SqlDecimal))
    Dim columnModifiedDate As DataColumn = _
      table.Columns.Add("ModifiedDate", GetType(SqlDateTime))

    ' Open a connection to SQL Server and fill the DataTable
    ' with data from the Sales.SalesOrderDetail table
    ' in the AdventureWorks sample database.
    Using connection As New SqlConnection(connectionString)

        Dim queryString As String = _
           "SELECT TOP 5 SalesOrderID, UnitPrice, LineTotal, ModifiedDate " _
           & "FROM Sales.SalesOrderDetail WHERE LineTotal < @LineTotal"

        ' Create the SqlCommand.
        Dim command As SqlCommand = New SqlCommand(queryString, connection)

        ' Create the SqlParameter and assign a value.
        Dim parameter As SqlParameter = _
           New SqlParameter("@LineTotal", SqlDbType.Decimal)
        parameter.Value = 1.5
        command.Parameters.Add(parameter)

        ' Open the connection and load the data.
        connection.Open()
        Dim reader As SqlDataReader = _
           command.ExecuteReader(CommandBehavior.CloseConnection)
        table.Load(reader)

        ' Close the SqlDataReader
        reader.Close()
    End Using

    ' Display the SqlType of each column.
    Dim column As DataColumn
    Console.WriteLine("Data Types:")
    For Each column In table.Columns
        Console.WriteLine(" {0} -- {1}", _
        column.ColumnName, column.DataType.UnderlyingSystemType)
    Next column

    ' Display the value for each row.
    Dim row As DataRow
    Console.WriteLine("Values:")
    For Each row In table.Rows
        Console.Write(" {0}, ", row("SalesOrderID"))
        Console.Write(" {0}, ", row("UnitPrice"))
        Console.Write(" {0}, ", row("LineTotal"))
        Console.Write(" {0} ", row("ModifiedDate"))
        Console.WriteLine()
    Next row
End Sub
```

See also

- SQL Server Data Type Mappings

- Configuring Parameters and Parameter Data Types
- ADO.NET Overview

Handling Null Values

4/26/2022 • 9 minutes to read • Edit Online

A null value in a relational database is used when the value in a column is unknown or missing. A null is neither an empty string (for character or datetime data types) nor a zero value (for numeric data types). The ANSI SQL-92 specification states that a null must be the same for all data types, so that all nulls are handled consistently. The System.Data.SqlTypes namespace provides null semantics by implementing the INullable interface. Each of the data types in System.Data.SqlTypes has its own `IsNull` property and a `Null` value that can be assigned to an instance of that data type.

> **NOTE**
>
> The .NET Framework version 2.0 introduced support for nullable value types, which allow programmers to extend a value type to represent all values of the underlying type. These CLR nullable value types represent an instance of the Nullable structure. This capability is especially useful when value types are boxed and unboxed, providing enhanced compatibility with object types. CLR nullable value types are not intended for storage of database nulls because an ANSI SQL null does not behave the same way as a `null` reference (or `Nothing` in Visual Basic). For working with database ANSI SQL null values, use System.Data.SqlTypes nulls rather than Nullable. For more information on working with CLR value nullable types in Visual Basic see Nullable Value Types, and for C# see Nullable value types.

Nulls and Three-Valued Logic

Allowing null values in column definitions introduces three-valued logic into your application. A comparison can evaluate to one of three conditions:

- True
- False
- Unknown

Because null is considered to be unknown, two null values compared to each other are not considered to be equal. In expressions using arithmetic operators, if any of the operands is null, the result is null as well.

Nulls and SqlBoolean

Comparison between any System.Data.SqlTypes will return a SqlBoolean. The `IsNull` function for each `SqlType` returns a SqlBoolean and can be used to check for null values. The following truth tables show how the AND, OR, and NOT operators function in the presence of a null value. (T=true, F=false, and U=unknown, or null.)

& (AND)	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

| (OR)	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

~(NOT)	
T	F
U	U
F	T

Understanding the ANSI_NULLS Option

System.Data.SqlTypes provides the same semantics as when the ANSI_NULLS option is set on in SQL Server. All arithmetic operators (+, -, *, /, %), bitwise operators (~, &, |), and most functions return null if any of the operands or arguments is null, except for the property `IsNull`.

The ANSI SQL-92 standard does not support *columnName* = NULL in a WHERE clause. In SQL Server, the ANSI_NULLS option controls both default nullability in the database and evaluation of comparisons against null values. If ANSI_NULLS is turned on (the default), the IS NULL operator must be used in expressions when testing for null values. For example, the following comparison always yields unknown when ANSI_NULLS is on:

```
colname > NULL
```

Comparison to a variable containing a null value also yields unknown:

```
colname > @MyVariable
```

Use the IS NULL or IS NOT NULL predicate to test for a null value. This can add complexity to the WHERE clause. For example, the TerritoryID column in the AdventureWorks Customer table allows null values. If a SELECT statement is to test for null values in addition to others, it must include an IS NULL predicate:

```
SELECT CustomerID, AccountNumber, TerritoryID
FROM AdventureWorks.Sales.Customer
WHERE TerritoryID IN (1, 2, 3)
   OR TerritoryID IS NULL
```

If you set ANSI_NULLS off in SQL Server, you can create expressions that use the equality operator to compare to null. However, you can't prevent different connections from setting null options for that connection. Using IS NULL to test for null values always works, regardless of the ANSI_NULLS settings for a connection.

Setting ANSI_NULLS off is not supported in a `DataSet`, which always follows the ANSI SQL-92 standard for handling null values in System.Data.SqlTypes.

Assigning Null Values

Null values are special, and their storage and assignment semantics differ across different type systems and storage systems. A `Dataset` is designed to be used with different type and storage systems.

This section describes the null semantics for assigning null values to a DataColumn in a DataRow across the

different type systems.

`DBNull.Value`
This assignment is valid for a `DataColumn` of any type. If the type implements `INullable`, `DBNull.Value` is coerced into the appropriate strongly typed Null value.

`SqlType.Null`
All System.Data.SqlTypes data types implement `INullable`. If the strongly typed null value can be converted into the column's data type using implicit cast operators, the assignment should go through. Otherwise an invalid cast exception is thrown.

`null`
If 'null' is a legal value for the given `DataColumn` data type, it is coerced into the appropriate `DbNull.Value` or `Null` associated with the `INullable` type (`SqlType.Null`)

`derivedUdt.Null`
For UDT columns, nulls are always stored based on the type associated with the `DataColumn`. Consider the case of a UDT associated with a `DataColumn` that does not implement `INullable` while its sub-class does. In this case, if a strongly typed null value associated with the derived class is assigned, it is stored as an untyped `DbNull.Value`, because null storage is always consistent with the DataColumn's data type.

> **NOTE**
> The `Nullable<T>` or Nullable structure is not currently supported in the `DataSet`.

The default value for any System.Data.SqlTypes instance is null.

Nulls in System.Data.SqlTypes are type-specific and cannot be represented by a single value, such as `DbNull`. Use the `IsNull` property to check for nulls.

Null values can be assigned to a DataColumn as shown in the following code example. You can directly assign null values to `SqlTypes` variables without triggering an exception.

Example

The following code example creates a DataTable with two columns defined as SqlInt32 and SqlString. The code adds one row of known values, one row of null values and then iterates through the DataTable, assigning the values to variables and displaying the output in the console window.

```
static private void WorkWithSqlNulls()
{
    DataTable table = new DataTable();

    // Specify the SqlType for each column.
    DataColumn idColumn =
        table.Columns.Add("ID", typeof(SqlInt32));
    DataColumn descColumn =
        table.Columns.Add("Description", typeof(SqlString));

    // Add some data.
    DataRow nRow = table.NewRow();
    nRow["ID"] = 123;
    nRow["Description"] = "Side Mirror";
    table.Rows.Add(nRow);

    // Add null values.
    nRow = table.NewRow();
    nRow["ID"] = SqlInt32.Null;
    nRow["Description"] = SqlString.Null;
    table.Rows.Add(nRow);

    // Initialize variables to use when
    // extracting the data.
    SqlBoolean isColumnNull = false;
    SqlInt32 idValue = SqlInt32.Zero;
    SqlString descriptionValue = SqlString.Null;

    // Iterate through the DataTable and display the values.
    foreach (DataRow row in table.Rows)
    {
        // Assign values to variables. Note that you
        // do not have to test for null values.
        idValue = (SqlInt32)row["ID"];
        descriptionValue = (SqlString)row["Description"];

        // Test for null value in ID column.
        isColumnNull = idValue.IsNull;

        // Display variable values in console window.
        Console.Write("isColumnNull={0}, ID={1}, Description={2}",
            isColumnNull, idValue, descriptionValue);
        Console.WriteLine();
    }
```

```
Private Sub WorkWithSqlNulls()
    Dim table As New DataTable()

    ' Specify the SqlType for each column.
    Dim idColumn As DataColumn = _
      table.Columns.Add("ID", GetType(SqlInt32))
    Dim descColumn As DataColumn = _
      table.Columns.Add("Description", GetType(SqlString))

    ' Add some data.
    Dim row As DataRow = table.NewRow()
    row("ID") = 123
    row("Description") = "Side Mirror"
    table.Rows.Add(row)

    ' Add null values.
    row = table.NewRow()
    row("ID") = SqlInt32.Null
    row("Description") = SqlString.Null
    table.Rows.Add(row)

    ' Initialize variables to use when
    ' extracting the data.
    Dim isColumnNull As SqlBoolean = False
    Dim idValue As SqlInt32 = SqlInt32.Zero
    Dim descriptionValue As SqlString = SqlString.Null

    ' Iterate through the DataTable and display the values.
    For Each row In table.Rows
        ' Assign values to variables. Note that you
        ' do not have to test for null values.
        idValue = CType(row("ID"), SqlInt32)
        descriptionValue = CType(row("Description"), SqlString)

        ' Test for null value with ID column
        isColumnNull = idValue.IsNull

        ' Display variable values in console window.
        Console.Write("isColumnNull={0}, ID={1}, Description={2}", _
          isColumnNull, idValue, descriptionValue)
        Console.WriteLine()
    Next row
End Sub
```

This example displays the following results:

```
isColumnNull=False, ID=123, Description=Side Mirror
isColumnNull=True, ID=Null, Description=Null
```

Multiple Column (Row) Assignment

`DataTable.Add`, `DataTable.LoadDataRow`, or other APIs that accept an ItemArray that gets mapped to a row, map 'null' to the DataColumn's default value. If an object in the array contains `DbNull.Value` or its strongly typed counterpart, the same rules as described above are applied.

In addition, the following rules apply for an instance of `DataRow.["columnName"]` null assignments:

1. The *default* value is `DbNull.Value` for all except the strongly typed null columns where it is the appropriate strongly typed null value.
2. Null values are never written out during serialization to XML files (as in "xsi:nil").
3. All non-null values, including defaults, are always written out while serializing to XML. This is unlike

XSD/XML semantics where a null value (xsi:nil) is explicit and the default value is implicit (if not present in XML, a validating parser can get it from an associated XSD schema). The opposite is true for a `DataTable`: a null value is implicit and the default value is explicit.

4. All missing column values for rows read from XML input are assigned NULL. Rows created using NewRow or similar methods are assigned the DataColumn's default value.

5. The IsNull method returns `true` for both `DbNull.Value` and `INullable.Null`.

Comparing Null Values with SqlTypes and CLR Types

When comparing null values, it is important to understand the difference between the way the `Equals` method evaluates null values in System.Data.SqlTypes as compared with the way it works with CLR types. All of the System.Data.SqlTypes `Equals` methods use database semantics for evaluating null values: if either or both of the values is null, the comparison yields null. On the other hand, using the CLR `Equals` method on two System.Data.SqlTypes will yield true if both are null. This reflects the difference between using an instance method such as the CLR `String.Equals` method, and using the static/shared method, `SqlString.Equals`.

The following example demonstrates the difference in results between the `SqlString.Equals` method and the `String.Equals` method when each is passed a pair of null values and then a pair of empty strings.

```
    private static void CompareNulls()
    {
        // Create two new null strings.
        SqlString a = new SqlString();
        SqlString b = new SqlString();

        // Compare nulls using static/shared SqlString.Equals.
        Console.WriteLine("SqlString.Equals shared/static method:");
        Console.WriteLine("  Two nulls={0}", SqlStringEquals(a, b));

        // Compare nulls using instance method String.Equals.
        Console.WriteLine();
        Console.WriteLine("String.Equals instance method:");
        Console.WriteLine("  Two nulls={0}", StringEquals(a, b));

        // Make them empty strings.
        a = "";
        b = "";

        // When comparing two empty strings (""), both the shared/static and
        // the instance Equals methods evaluate to true.
        Console.WriteLine();
        Console.WriteLine("SqlString.Equals shared/static method:");
        Console.WriteLine("  Two empty strings={0}", SqlStringEquals(a, b));

        Console.WriteLine();
        Console.WriteLine("String.Equals instance method:");
        Console.WriteLine("  Two empty strings={0}", StringEquals(a, b));
    }

    private static string SqlStringEquals(SqlString string1, SqlString string2)
    {
        // SqlString.Equals uses database semantics for evaluating nulls.
        string returnValue = SqlString.Equals(string1, string2).ToString();
        return returnValue;
    }

    private static string StringEquals(SqlString string1, SqlString string2)
    {
        // String.Equals uses CLR type semantics for evaluating nulls.
        string returnValue = string1.Equals(string2).ToString();
        return returnValue;
    }
}
```

```
Private Sub CompareNulls()
    ' Create two new null strings.
    Dim a As New SqlString
    Dim b As New SqlString

    ' Compare nulls using static/shared SqlString.Equals.
    Console.WriteLine("SqlString.Equals shared/static method:")
    Console.WriteLine("  Two nulls={0}", SqlStringEquals(a, b))

    ' Compare nulls using instance method String.Equals.
    Console.WriteLine()
    Console.WriteLine("String.Equals instance method:")
    Console.WriteLine("  Two nulls={0}", StringEquals(a, b))

    ' Make them empty strings.
    a = ""
    b = ""

    ' When comparing two empty strings (""), both the shared/static and
    ' the instance Equals methods evaluate to true.
    Console.WriteLine()
    Console.WriteLine("SqlString.Equals shared/static method:")
    Console.WriteLine("  Two empty strings={0}", SqlStringEquals(a, b))

    Console.WriteLine()
    Console.WriteLine("String.Equals instance method:")
    Console.WriteLine("  Two empty strings={0}", StringEquals(a, b))
End Sub

Private Function SqlStringEquals(ByVal string1 As SqlString, _
    ByVal string2 As SqlString) As String

    ' SqlString.Equals uses database semantics for evaluating nulls.
    Dim returnValue As String = SqlString.Equals(string1, string2).ToString()
    Return returnValue
End Function

Private Function StringEquals(ByVal string1 As SqlString, _
    ByVal string2 As SqlString) As String

    ' String.Equals uses CLR type semantics for evaluating nulls.
    Dim returnValue As String = string1.Equals(string2).ToString()
    Return returnValue
End Function
```

The code produces the following output:

```
SqlString.Equals shared/static method:
  Two nulls=Null

String.Equals instance method:
  Two nulls=True

SqlString.Equals shared/static method:
  Two empty strings=True

String.Equals instance method:
  Two empty strings=True
```

See also

- SQL Server Data Types and ADO.NET
- ADO.NET Overview

Comparing GUID and uniqueidentifier Values

4/26/2022 • 2 minutes to read • Edit Online

The globally unique identifier (GUID) data type in SQL Server is represented by the `uniqueidentifier` data type, which stores a 16-byte binary value. A GUID is a binary number, and its main use is as an identifier that must be unique in a network that has many computers at many sites. GUIDs can be generated by calling the Transact-SQL NEWID function, and is guaranteed to be unique throughout the world. For more information, see uniqueidentifier (Transact-SQL).

Working with SqlGuid Values

Because GUIDs values are long and obscure, they are not meaningful for users. If randomly generated GUIDs are used for key values and you insert a lot of rows, you get random I/O into your indexes, which can negatively impact performance. GUIDs are also relatively large when compared to other data types. In general we recommend using GUIDs only for very narrow scenarios for which no other data type is suitable.

Comparing GUID Values

Comparison operators can be used with `uniqueidentifier` values. However, ordering is not implemented by comparing the bit patterns of the two values. The only operations that are allowed against a `uniqueidentifier` value are comparisons (=, <>, <, >, <=, >=) and checking for NULL (IS NULL and IS NOT NULL). No other arithmetic operators are allowed.

Both Guid and SqlGuid have a `CompareTo` method for comparing different GUID values. However, `System.Guid.CompareTo` and `SqlTypes.SqlGuid.CompareTo` are implemented differently. SqlGuid implements `CompareTo` using SQL Server behavior, in the last six bytes of a value are most significant. Guid evaluates all 16 bytes. The following example demonstrates this behavioral difference. The first section of code displays unsorted Guid values, and the second section of code shows the sorted Guid values. The third section shows the sorted SqlGuid values. The output is displayed beneath the code listing.

```
private static void WorkWithGuids()
{
    // Create an ArrayList and fill it with Guid values.
    ArrayList guidList = new ArrayList();
    guidList.Add(new Guid("3AAAAAAA-BBBB-CCCC-DDDD-2EEEEEEEEEE"));
    guidList.Add(new Guid("2AAAAAAA-BBBB-CCCC-DDDD-1EEEEEEEEEE"));
    guidList.Add(new Guid("1AAAAAAA-BBBB-CCCC-DDDD-3EEEEEEEEEE"));

    // Display the unsorted Guid values.
    Console.WriteLine("Unsorted Guids:");
    foreach (Guid guidValue in guidList)
    {
        Console.WriteLine(" {0}", guidValue);
    }
    Console.WriteLine("");

    // Sort the Guids.
    guidList.Sort();

    // Display the sorted Guid values.
    Console.WriteLine("Sorted Guids:");
    foreach (Guid guidSorted in guidList)
    {
        Console.WriteLine(" {0}", guidSorted);
    }
    Console.WriteLine("");

    // Create an ArrayList of SqlGuids.
    ArrayList sqlGuidList = new ArrayList();
    sqlGuidList.Add(new SqlGuid("3AAAAAAA-BBBB-CCCC-DDDD-2EEEEEEEEEE"));
    sqlGuidList.Add(new SqlGuid("2AAAAAAA-BBBB-CCCC-DDDD-1EEEEEEEEEE"));
    sqlGuidList.Add(new SqlGuid("1AAAAAAA-BBBB-CCCC-DDDD-3EEEEEEEEEE"));

    // Sort the SqlGuids. The unsorted SqlGuids are in the same order
    // as the unsorted Guid values.
    sqlGuidList.Sort();

    // Display the sorted SqlGuids. The sorted SqlGuid values are ordered
    // differently than the Guid values.
    Console.WriteLine("Sorted SqlGuids:");
    foreach (SqlGuid sqlGuidValue in sqlGuidList)
    {
        Console.WriteLine(" {0}", sqlGuidValue);
    }
}
```

```
Private Sub WorkWithGuids()

    ' Create an ArrayList and fill it with Guid values.
    Dim guidList As New ArrayList()
    guidList.Add(New Guid("3AAAAAAA-BBBB-CCCC-DDDD-2EEEEEEEEEE"))
    guidList.Add(New Guid("2AAAAAAA-BBBB-CCCC-DDDD-1EEEEEEEEEE"))
    guidList.Add(New Guid("1AAAAAAA-BBBB-CCCC-DDDD-3EEEEEEEEEE"))

    ' Display the unsorted Guid values.
    Console.WriteLine("Unsorted Guids:")
    For Each guidValue As Guid In guidList
        Console.WriteLine("{0}", guidValue)
    Next
    Console.WriteLine()

    ' Sort the Guids.
    guidList.Sort()

    ' Display the sorted Guid values.

    Console.WriteLine("Sorted Guids:")
    For Each guidSorted As Guid In guidList
        Console.WriteLine("{0}", guidSorted)
    Next
    Console.WriteLine()

    ' Create an ArrayList of SqlGuids.
    Dim sqlGuidList As New ArrayList()
    sqlGuidList.Add(New SqlGuid("3AAAAAAA-BBBB-CCCC-DDDD-2EEEEEEEEEE"))
    sqlGuidList.Add(New SqlGuid("2AAAAAAA-BBBB-CCCC-DDDD-1EEEEEEEEEE"))
    sqlGuidList.Add(New SqlGuid("1AAAAAAA-BBBB-CCCC-DDDD-3EEEEEEEEEE"))

    ' Sort the SqlGuids. The unsorted SqlGuids are in the same order
    ' as the unsorted Guid values.
    sqlGuidList.Sort()

    ' Display the sorted SqlGuids. The sorted SqlGuid values are
    ' ordered differently than the Guid values.
    Console.WriteLine("Sorted SqlGuids:")
    For Each sqlGuidValue As SqlGuid In sqlGuidList
        Console.WriteLine("{0}", sqlGuidValue)
    Next
End Sub
```

This example produces the following results.

```
Unsorted Guids:
3aaaaaaa-bbbb-cccc-dddd-2eeeeeeeeee
2aaaaaaa-bbbb-cccc-dddd-1eeeeeeeeee
1aaaaaaa-bbbb-cccc-dddd-3eeeeeeeeee

Sorted Guids:
1aaaaaaa-bbbb-cccc-dddd-3eeeeeeeeee
2aaaaaaa-bbbb-cccc-dddd-1eeeeeeeeee
3aaaaaaa-bbbb-cccc-dddd-2eeeeeeeeee

Sorted SqlGuids:
2aaaaaaa-bbbb-cccc-dddd-1eeeeeeeeee
3aaaaaaa-bbbb-cccc-dddd-2eeeeeeeeee
1aaaaaaa-bbbb-cccc-dddd-3eeeeeeeeee
```

See also

- SQL Server Data Types and ADO.NET

- ADO.NET Overview

Date and Time Data

4/26/2022 • 9 minutes to read • Edit Online

SQL Server 2008 introduces new data types for handling date and time information. The new data types include separate types for date and time, and expanded data types with greater range, precision, and time-zone awareness. Starting with the .NET Framework version 3.5 Service Pack (SP) 1, the .NET Framework Data Provider for SQL Server (System.Data.SqlClient) provides full support for all the new features of the SQL Server 2008 Database Engine. You must install the .NET Framework 3.5 SP1 (or later) to use these new features with SqlClient.

Versions of SQL Server earlier than SQL Server 2008 only had two data types for working with date and time values: `datetime` and `smalldatetime`. Both of these data types contain both the date value and a time value, which makes it difficult to work with only date or only time values. Also, these data types only support dates that occur after the introduction of the Gregorian calendar in England in 1753. Another limitation is that these older data types are not time-zone aware, which makes it difficult to work with data that originates from multiple time zones.

For more information about date and time types in SQL Server, see Date and Time Data Types and Functions.

Date/Time Data Types Introduced in SQL Server 2008

The following table describes the new date and time data types.

SQL SERVER DATA TYPE	DESCRIPTION
`date`	The `date` data type has a range of January 1, 01 through December 31, 9999 with an accuracy of 1 day. The default value is January 1, 1900. The storage size is 3 bytes.
`time`	The `time` data type stores time values only, based on a 24-hour clock. The `time` data type has a range of 00:00:00.0000000 through 23:59:59.9999999 with an accuracy of 100 nanoseconds. The default value is 00:00:00.0000000 (midnight). The `time` data type supports user-defined fractional second precision, and the storage size varies from 3 to 6 bytes, based on the precision specified.
`datetime2`	The `datetime2` data type combines the range and precision of the `date` and `time` data types into a single data type. The default values and string literal formats are the same as those defined in the `date` and `time` data types.

SQL SERVER DATA TYPE	DESCRIPTION
`datetimeoffset`	The `datetimeoffset` data type has all the features of `datetime2` with an additional time zone offset. The time zone offset is represented as [+|-] HH:MM. HH is 2 digits ranging from 00 to 14 that represent the number of hours in the time zone offset. MM is 2 digits ranging from 00 to 59 that represent the number of additional minutes in the time zone offset. Time formats are supported to 100 nanoseconds. The mandatory + or - sign indicates whether the time zone offset is added or subtracted from UTC (Universal Time Coordinate or Greenwich Mean Time) to obtain the local time.

NOTE

For more information about using the `Type System Version` keyword, see ConnectionString.

Date Format and Date Order

How SQL Server parses date and time values depends not only on the type system version and server version, but also on the server's default language and format settings. A date string that works for the date formats of one language might be unrecognizable if the query is executed by a connection that uses a different language and date format setting.

The Transact-SQL SET LANGUAGE statement implicitly sets the DATEFORMAT that determines the order of the date parts. You can use the SET DATEFORMAT Transact-SQL statement on a connection to disambiguate date values by ordering the date parts in MDY, DMY, YMD, YDM, MYD, or DYM order.

If you do not specify any DATEFORMAT for the connection, SQL Server uses the default language associated with the connection. For example, a date string of '01/02/03' would be interpreted as MDY (January 2, 2003) on a server with a language setting of United States English, and as DMY (February 1, 2003) on a server with a language setting of British English. The year is determined by using SQL Server's cutoff year rule, which defines the cutoff date for assigning the century value. For more information, see two digit year cutoff Option.

NOTE

The YDM date format is not supported when converting from a string format to `date`, `time`, `datetime2`, or `datetimeoffset`.

For more information about how SQL Server interprets date and time data, see Using Date and Time Data.

Date/Time Data Types and Parameters

The following enumerations have been added to SqlDbType to support the new date and time data types.

- `SqlDbType.Date`
- `SqlDbType.Time`
- `SqlDbType.DateTime2`
- `SqlDbType.DateTimeOffSet`

You can specify the data type of a SqlParameter by using one of the preceding SqlDbType enumerations.

> **NOTE**
> You cannot set the `DbType` property of a `SqlParameter` to `SqlDbType.Date`.

You can also specify the type of a SqlParameter generically by setting the DbType property of a `SqlParameter` object to a particular DbType enumeration value. The following enumeration values have been added to DbType to support the `datetime2` and `datetimeoffset` data types:

- DbType.DateTime2
- DbType.DateTimeOffset

These new enumerations supplement the `Date`, `Time`, and `DateTime` enumerations, which existed in earlier versions of the .NET Framework.

The .NET Framework data provider type of a parameter object is inferred from the .NET Framework type of the value of the parameter object, or from the `DbType` of the parameter object. No new System.Data.SqlTypes data types have been introduced to support the new date and time data types. The following table describes the mappings between the SQL Server 2008 date and time data types and the CLR data types.

SQL SERVER DATA TYPE	.NET FRAMEWORK TYPE	SYSTEM.DATA.SQLDBTYPE	SYSTEM.DATA.DBTYPE
date	System.DateTime	Date	Date
time	System.TimeSpan	Time	Time
datetime2	System.DateTime	DateTime2	DateTime2
datetimeoffset	System.DateTimeOffset	DateTimeOffset	DateTimeOffset
datetime	System.DateTime	DateTime	DateTime
smalldatetime	System.DateTime	DateTime	DateTime

SqlParameter Properties

The following table describes `SqlParameter` properties that are relevant to date and time data types.

PROPERTY	DESCRIPTION
IsNullable	Gets or sets whether a value is nullable. When you send a null parameter value to the server, you must specify DBNull, rather than `null` (`Nothing` in Visual Basic). For more information about database nulls, see Handling Null Values.
Precision	Gets or sets the maximum number of digits used to represent the value. This setting is ignored for date and time data types.
Scale	Gets or sets the number of decimal places to which the time portion of the value is resolved for `Time`, `DateTime2`,and `DateTimeOffset`. The default value is 0, which means that the actual scale is inferred from the value and sent to the server.

PROPERTY	DESCRIPTION
Size	Ignored for date and time data types.
Value	Gets or sets the parameter value.
SqlValue	Gets or sets the parameter value.

> **NOTE**
> Time values that are less than zero or greater than or equal to 24 hours will throw an ArgumentException.

Creating Parameters

You can create a SqlParameter object by using its constructor, or by adding it to a SqlCommandParameters collection by calling the `Add` method of the SqlParameterCollection. The `Add` method will take as input either constructor arguments or an existing parameter object.

The next sections in this topic provide examples of how to specify date and time parameters. For additional examples of working with parameters, see Configuring Parameters and Parameter Data Types and DataAdapter Parameters.

Date Example

The following code fragment demonstrates how to specify a `date` parameter.

```
SqlParameter parameter = new SqlParameter();
parameter.ParameterName = "@Date";
parameter.SqlDbType = SqlDbType.Date;
parameter.Value = "2007/12/1";
```

```
Dim parameter As New SqlParameter()
parameter.ParameterName = "@Date"
parameter.SqlDbType = SqlDbType.Date
parameter.Value = "2007/12/1"
```

Time Example

The following code fragment demonstrates how to specify a `time` parameter.

```
SqlParameter parameter = new SqlParameter();
parameter.ParameterName = "@time";
parameter.SqlDbType = SqlDbType.Time;
parameter.Value = DateTime.Parse("23:59:59").TimeOfDay;
```

```
Dim parameter As New SqlParameter()
parameter.ParameterName = "@Time"
parameter.SqlDbType = SqlDbType.Time
parameter.Value = DateTime.Parse("23:59:59").TimeOfDay;
```

Datetime2 Example

The following code fragment demonstrates how to specify a `datetime2` parameter with both the date and time parts.

```
SqlParameter parameter = new SqlParameter();
parameter.ParameterName = "@Datetime2";
parameter.SqlDbType = SqlDbType.DateTime2;
parameter.Value = DateTime.Parse("1666-09-02 1:00:00");
```

```
Dim parameter As New SqlParameter()
parameter.ParameterName = "@Datetime2"
parameter.SqlDbType = SqlDbType.DateTime2
parameter.Value = DateTime.Parse("1666-09-02 1:00:00");
```

DateTimeOffSet Example

The following code fragment demonstrates how to specify a `DateTimeOffSet` parameter with a date, a time, and a time zone offset of 0.

```
SqlParameter parameter = new SqlParameter();
parameter.ParameterName = "@DateTimeOffSet";
parameter.SqlDbType = SqlDbType.DateTimeOffSet;
parameter.Value = DateTimeOffset.Parse("1666-09-02 1:00:00+0");
```

```
Dim parameter As New SqlParameter()
parameter.ParameterName = "@DateTimeOffSet"
parameter.SqlDbType = SqlDbType.DateTimeOffSet
parameter.Value = DateTimeOffset.Parse("1666-09-02 1:00:00+0");
```

AddWithValue

You can also supply parameters by using the `AddWithValue` method of a SqlCommand, as shown in the following code fragment. However, the `AddWithValue` method does not allow you to specify the DbType or SqlDbType for the parameter.

```
command.Parameters.AddWithValue(
    "@date", DateTimeOffset.Parse("16660902"));
```

```
command.Parameters.AddWithValue( _
    "@date", DateTimeOffset.Parse("16660902"))
```

The `@date` parameter could map to a `date`, `datetime`, or `datetime2` data type on the server. When working with the new `datetime` data types, you must explicitly set the parameter's SqlDbType property to the data type of the instance. Using Variant or implicitly supplying parameter values can cause problems with backward compatibility with the `datetime` and `smalldatetime` data types.

The following table shows which `SqlDbTypes` are inferred from which CLR types:

CLR TYPE	INFERRED SQLDBTYPE
DateTime	SqlDbType.DateTime
TimeSpan	SqlDbType.Time
DateTimeOffset	SqlDbType.DateTimeOffset

Retrieving Date and Time Data

The following table describes methods that are used to retrieve SQL Server 2008 date and time values.

SQLCLIENT METHOD	DESCRIPTION
GetDateTime	Retrieves the specified column value as a DateTime structure.
GetDateTimeOffset	Retrieves the specified column value as a DateTimeOffset structure.
GetProviderSpecificFieldType	Returns the type that is the underlying provider-specific type for the field. Returns the same types as `GetFieldType` for new date and time types.
GetProviderSpecificValue	Retrieves the value of the specified column. Returns the same types as `GetValue` for the new date and time types.
GetProviderSpecificValues	Retrieves the values in the specified array.
GetSqlString	Retrieves the column value as a SqlString. An InvalidCastException occurs if the data cannot be expressed as a `SqlString`.
GetSqlValue	Retrieves column data as its default `SqlDbType`. Returns the same types as `GetValue` for the new date and time types.
GetSqlValues	Retrieves the values in the specified array.
GetString	Retrieves the column value as a string if the Type System Version is set to SQL Server 2005. An InvalidCastException occurs if the data cannot be expressed as a string.
GetTimeSpan	Retrieves the specified column value as a TimeSpan structure.
GetValue	Retrieves the specified column value as its underlying CLR type.
GetValues	Retrieves column values in an array.
GetSchemaTable	Returns a DataTable that describes the metadata of the result set.

NOTE

The new date and time `SqlDbTypes` are not supported for code that is executing in-process in SQL Server. An exception will be raised if one of these types is passed to the server.

Specifying Date and Time Values as Literals

You can specify date and time data types by using a variety of different literal string formats, which SQL Server then evaluates at run time, converting them to internal date/time structures. SQL Server recognizes date and time data that is enclosed in single quotation marks ('). The following examples demonstrate some formats:

- Alphabetic date formats, such as `'October 15, 2006'`.
- Numeric date formats, such as `'10/15/2006'`.
- Unseparated string formats, such as `'20061015'`, which would be interpreted as October 15, 2006 if you are using the ISO standard date format.

Time values that are less than zero or greater than or equal to 24 hours will throw an ArgumentException.

SQL Server docs resources

For more information about working with date and time values in SQL Server, see the following articles.

ARTICLE	DESCRIPTION
Date and Time Data Types and Functions (Transact-SQL)	Provides an overview of all Transact-SQL date and time data types and functions.
Using Date and Time Data	Provides information about the date and time data types and functions, and examples of using them.
Data Types (Transact-SQL)	Describes system data types in SQL Server.

See also

- SQL Server Data Type Mappings
- Configuring Parameters and Parameter Data Types
- SQL Server Data Types and ADO.NET
- ADO.NET Overview

Large UDTs

4/26/2022 • 3 minutes to read • Edit Online

User-defined types (UDTs) allow a developer to extend the server's scalar type system by storing common language runtime (CLR) objects in a SQL Server database. UDTs can contain multiple elements and can have behaviors, unlike the traditional alias data types, which consist of a single SQL Server system data type.

> **NOTE**
>
> You must install the .NET Framework 3.5 SP1 (or later) to take advantage of the enhanced SqlClient support for large UDTs.

Previously, UDTs were restricted to a maximum size of 8 kilobytes. In SQL Server 2008, this restriction has been removed for UDTs that have a format of UserDefined.

For the complete documentation for user-defined types, see CLR User-Defined Types.

Retrieving UDT Schemas Using GetSchema

The GetSchema method of SqlConnection returns database schema information in a DataTable. For more information, see SQL Server Schema Collections.

GetSchemaTable Column Values for UDTs

The GetSchemaTable method of a SqlDataReader returns a DataTable that describes column metadata. The following table describes the differences in the column metadata for large UDTs between SQL Server 2005 and SQL Server 2008.

SQLDATAREADER COLUMN	SQL SERVER 2005	SQL SERVER 2008 AND LATER
`ColumnSize`	Varies	Varies
`NumericPrecision`	255	255
`NumericScale`	255	255
`DataType`	`Byte[]`	UDT instance
`ProviderSpecificDataType`	`SqlTypes.SqlBinary`	UDT instance
`ProviderType`	21 (`SqlDbType.VarBinary`)	29 (`SqlDbType.Udt`)
`NonVersionedProviderType`	29 (`SqlDbType.Udt`)	29 (`SqlDbType.Udt`)
`DataTypeName`	`SqlDbType.VarBinary`	The three part name specified as *Database.SchemaName.TypeName*.
`IsLong`	Varies	Varies

SqlDataReader Considerations

The SqlDataReader has been extended beginning in SQL Server 2008 to support retrieving large UDT values. How large UDT values are processed by a SqlDataReader depends on the version of SQL Server you are using, as well as on the `Type System Version` specified in the connection string. For more information, see ConnectionString.

The following methods of SqlDataReader will return a SqlBinary instead of a UDT when the `Type System Version` is set to SQL Server 2005:

- GetProviderSpecificFieldType
- GetProviderSpecificValue
- GetProviderSpecificValues
- GetSqlValue
- GetSqlValues

The following methods will return an array of `Byte[]` instead of a UDT when the `Type System Version` is set to SQL Server 2005:

- GetValue
- GetValues

Note that no conversions are made for the current version of ADO.NET.

Specifying SqlParameters

The following SqlParameter properties have been extended to work with large UDTs.

SQLPARAMETER PROPERTY	DESCRIPTION
Value	Gets or sets an object that represents the value of the parameter. The default is null. The property can be `SqlBinary`, `Byte[]`, or a managed object.
SqlValue	Gets or sets an object that represents the value of the parameter. The default is null. The property can be `SqlBinary`, `Byte[]`, or a managed object.
Size	Gets or sets the size of the parameter value to resolve. The default value is 0. The property can be an integer that represents the size of the parameter value. For large UDTs, it can be the actual size of the UDT, or -1 for unknown.

Retrieving Data Example

The following code fragment demonstrates how to retrieve large UDT data. The `connectionString` variable assumes a valid connection to a SQL Server database and the `commandString` variable assumes a valid SELECT statement with the primary key column listed first.

```
using (SqlConnection connection = new SqlConnection(
    connectionString, commandString))
{
  connection.Open();
  SqlCommand command = new SqlCommand(commandString);
  SqlDataReader reader = command.ExecuteReader();
  while (reader.Read())
  {
    // Retrieve the value of the Primary Key column.
    int id = reader.GetInt32(0);

    // Retrieve the value of the UDT.
    LargeUDT udt = (LargeUDT)reader[1];

    // You can also use GetSqlValue and GetValue.
    // LargeUDT udt = (LargeUDT)reader.GetSqlValue(1);
    // LargeUDT udt = (LargeUDT)reader.GetValue(1);

    Console.WriteLine(
     "ID={0} LargeUDT={1}", id, udt);
  }
reader.close
}
```

```
Using connection As New SqlConnection( _
    connectionString, commandString)
    connection.Open()
    Dim command As New SqlCommand(commandString, connection)
    Dim reader As SqlDataReader
    reader = command.ExecuteReader

    While reader.Read()
      ' Retrieve the value of the Primary Key column.
      Dim id As Int32 = reader.GetInt32(0)

      ' Retrieve the value of the UDT.
      Dim udt As LargeUDT = CType(reader(1), LargeUDT)

     ' You can also use GetSqlValue and GetValue.
     ' Dim udt As LargeUDT = CType(reader.GetSqlValue(1), LargeUDT)
     ' Dim udt As LargeUDT = CType(reader.GetValue(1), LargeUDT)

      ' Print values.
      Console.WriteLine("ID={0} LargeUDT={1}", id, udt)
    End While
    reader.Close()
End Using
```

See also

- Configuring Parameters and Parameter Data Types
- Retrieving Database Schema Information
- SQL Server Data Type Mappings
- SQL Server Binary and Large-Value Data
- ADO.NET Overview

XML Data in SQL Server

4/26/2022 • 2 minutes to read • Edit Online

SQL Server exposes the functionality of SQLXML inside the .NET Framework. Developers can write applications that access XML data from an instance of SQL Server, bring the data into the .NET Framework environment, process the data, and send the updates back to SQL Server. XML data can be used in several ways in SQL Server, including data storage, and as parameter values for retrieving data. The **SqlXml** class in the .NET Framework provides the client-side support for working with data stored in an XML column within SQL Server. For more information, see SQLXML Managed Classes.

In This Section

SQL XML Column Values
Demonstrates how to retrieve and work with XML data retrieved from SQL Server.

Specifying XML Values as Parameters
Demonstrates how to pass XML data as a parameter to a command.

See also

- SQL Server and ADO.NET
- ADO.NET Overview

SQL XML Column Values

4/26/2022 • 3 minutes to read • Edit Online

SQL Server supports the `xml` data type, and developers can retrieve result sets including this type using standard behavior of the SqlCommand class. An `xml` column can be retrieved just as any column is retrieved (into a SqlDataReader, for example) but if you want to work with the content of the column as XML, you must use an XmlReader.

Example

The following console application selects two rows, each containing an `xml` column, from the **Sales.Store** table in the **AdventureWorks** database to a SqlDataReader instance. For each row, the value of the `xml` column is read using the GetSqlXml method of SqlDataReader. The value is stored in an XmlReader. Note that you must use GetSqlXml rather than the GetValue method if you want to set the contents to a SqlXml variable; GetValue returns the value of the `xml` column as a string.

> **NOTE**
>
> The **AdventureWorks** sample database is not installed by default when you install SQL Server. You can install it by running SQL Server Setup.

```
// Example assumes the following directives:
//     using System.Data.SqlClient;
//     using System.Xml;
//     using System.Data.SqlTypes;

static void GetXmlData(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        // The query includes two specific customers for simplicity's
        // sake. A more realistic approach would use a parameter
        // for the CustomerID criteria. The example selects two rows
        // in order to demonstrate reading first from one row to
        // another, then from one node to another within the xml column.
        string commandText =
            "SELECT Demographics from Sales.Store WHERE " +
            "CustomerID = 3 OR CustomerID = 4";

        SqlCommand commandSales = new SqlCommand(commandText, connection);

        SqlDataReader salesReaderData = commandSales.ExecuteReader();

        //  Multiple rows are returned by the SELECT, so each row
        //  is read and an XmlReader (an xml data type) is set to the
        //  value of its first (and only) column.
        int countRow = 1;
        while (salesReaderData.Read())
        //  Must use GetSqlXml here to get a SqlXml type.
        //  GetValue returns a string instead of SqlXml.
        {
            SqlXml salesXML =
                salesReaderData.GetSqlXml(0);
            XmlReader salesReaderXml = salesXML.CreateReader();
            Console.WriteLine("-----Row " + countRow + "-----");

            //  Move to the root.
            salesReaderXml.MoveToContent();

            //  We know each node type is either Element or Text.
            //  All elements within the root are string values.
            //  For this simple example, no elements are empty.
            while (salesReaderXml.Read())
            {
                if (salesReaderXml.NodeType == XmlNodeType.Element)
                {
                    string elementLocalName =
                        salesReaderXml.LocalName;
                    salesReaderXml.Read();
                    Console.WriteLine(elementLocalName + ": " +
                        salesReaderXml.Value);
                }
            }
            countRow = countRow + 1;
        }
    }
}
```

```
' Example assumes the following directives:
'    Imports System.Data.SqlClient
'    Imports System.Xml
'    Imports System.Data.SqlTypes

Private Sub GetXmlData(ByVal connectionString As String)
    Using connection As SqlConnection = New SqlConnection(connectionString)
        connection.Open()

        'The query includes two specific customers for simplicity's
        'sake. A more realistic approach would use a parameter
        'for the CustomerID criteria. The example selects two rows
        'in order to demonstrate reading first from one row to
        'another, then from one node to another within the xml
        'column.
        Dim commandText As String = _
         "SELECT Demographics from Sales.Store WHERE " & _
         "CustomerID = 3 OR CustomerID = 4"

        Dim commandSales As New SqlCommand(commandText, connection)

        Dim salesReaderData As SqlDataReader = commandSales.ExecuteReader()

        ' Multiple rows are returned by the SELECT, so each row
        ' is read and an XmlReader (an xml data type) is set to the
        ' value of its first (and only) column.
        Dim countRow As Integer = 1
        While salesReaderData.Read()
            ' Must use GetSqlXml here to get a SqlXml type.
            ' GetValue returns a string instead of SqlXml.
            Dim salesXML As SqlXml = _
             salesReaderData.GetSqlXml(0)
            Dim salesReaderXml As XmlReader = salesXML.CreateReader()

            Console.WriteLine("-----Row " & countRow & "-----")

            ' Move to the root.
            salesReaderXml.MoveToContent()

            ' We know each node type is either Element or Text.
            ' All elements within the root are string values.
            ' For this simple example, no elements
            ' are empty.
            While salesReaderXml.Read()
                If salesReaderXml.NodeType = XmlNodeType.Element Then
                    Dim elementLocalName As String = _
                     salesReaderXml.LocalName
                    salesReaderXml.Read()
                    Console.WriteLine(elementLocalName & ": " & _
                     salesReaderXml.Value)
                End If
            End While
            countRow = countRow + 1
        End While
    End Using
End Sub
```

See also

- SqlXml
- XML Data in SQL Server
- ADO.NET Overview

Specifying XML Values as Parameters

4/26/2022 • 2 minutes to read • Edit Online

If a query requires a parameter whose value is an XML string, developers can supply that value using an instance of the **SqlXml** data type. There really are no tricks; XML columns in SQL Server accept parameter values in exactly the same way as other data types.

Example

The following console application creates a new table in the **AdventureWorks** database. The new table includes a column named **SalesID** and an XML column named **SalesInfo**.

> **NOTE**
>
> The **AdventureWorks** sample database is not installed by default when you install SQL Server. You can install it by running SQL Server Setup.

The example prepares a SqlCommand object to insert a row in the new table. A saved file provides the XML data needed for the **SalesInfo** column.

To create the file needed for the example to run, create a new text file in the same folder as your project. Name the file MyTestStoreData.xml. Open the file in Notepad and copy and paste the following text:

```
<StoreSurvey xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey">
  <AnnualSales>300000</AnnualSales>
  <AnnualRevenue>30000</AnnualRevenue>
  <BankName>International Bank</BankName>
  <BusinessType>BM</BusinessType>
  <YearOpened>1970</YearOpened>
  <Specialty>Road</Specialty>
  <SquareFeet>7000</SquareFeet>
  <Brands>3</Brands>
  <Internet>T1</Internet>
  <NumberEmployees>2</NumberEmployees>
</StoreSurvey>
```

```
Imports System
Imports System.Data.SqlClient
Imports System.Data.SqlTypes
Imports System.Xml

Module Module1
    Sub Main()

        Using connection As SqlConnection = New SqlConnection(GetConnectionString())
        connection.Open()

        ' Create a sample table (dropping first if it already
        ' exists.)
        Dim commandNewTable As String = _
         "IF EXISTS (SELECT * FROM dbo.sysobjects " & _
         "WHERE id = object_id(N'[dbo].[XmlDataTypeSample]') " & _
         "AND OBJECTPROPERTY(id, N'IsUserTable') = 1) " & _
         "DROP TABLE [dbo].[XmlDataTypeSample];" & _
         "CREATE TABLE [dbo].[XmlDataTypeSample](" & _
         "[SalesID] [int] IDENTITY(1,1) NOT NULL, " & _
         "[SalesInfo] [xml])"

        Dim commandAdd As New _
         SqlCommand(commandNewTable, connection)
        commandAdd.ExecuteNonQuery()

        Dim commandText As String = _
         "INSERT INTO [dbo].[XmlDataTypeSample] " & _
           "([SalesInfo] ) " & _
           "VALUES(@xmlParameter )"

        Dim command As New SqlCommand(commandText, connection)

        ' Read the saved XML document as a
        ' SqlXml-data typed variable.
        Dim newXml As SqlXml = _
         New SqlXml(New XmlTextReader("MyTestStoreData.xml"))

        ' Supply the SqlXml value for the value of the parameter.
        command.Parameters.AddWithValue("@xmlParameter", newXml)

        Dim result As Integer = command.ExecuteNonQuery()
        Console.WriteLine(result & " row was added.")
        Console.WriteLine("Press Enter to continue.")
        Console.ReadLine()
    End Using
End Sub

    Private Function GetConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.
        Return "Data Source=(local);Integrated Security=SSPI;" & _
          "Initial Catalog=AdventureWorks"
    End Function
End Module
```

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Xml;
using System.Data.SqlTypes;

class Class1
{
    static void Main()
    {
        using (SqlConnection connection = new SqlConnection(GetConnectionString()))
       {
        connection.Open();
        //  Create a sample table (dropping first if it already
        //  exists.)

        string commandNewTable =
            "IF EXISTS (SELECT * FROM dbo.sysobjects " +
            "WHERE id = " +
                  "object_id(N'[dbo].[XmlDataTypeSample]') " +
            "AND OBJECTPROPERTY(id, N'IsUserTable') = 1) " +
            "DROP TABLE [dbo].[XmlDataTypeSample];" +
            "CREATE TABLE [dbo].[XmlDataTypeSample](" +
            "[SalesID] [int] IDENTITY(1,1) NOT NULL, " +
            "[SalesInfo] [xml])";
        SqlCommand commandAdd =
                   new SqlCommand(commandNewTable, connection);
        commandAdd.ExecuteNonQuery();
        string commandText =
            "INSERT INTO [dbo].[XmlDataTypeSample] " +
            "([SalesInfo] ) " +
            "VALUES(@xmlParameter )";
        SqlCommand command =
                  new SqlCommand(commandText, connection);

        //  Read the saved XML document as a
        //  SqlXml-data typed variable.
        SqlXml newXml =
            new SqlXml(new XmlTextReader("MyTestStoreData.xml"));

        //  Supply the SqlXml value for the value of the parameter.
        command.Parameters.AddWithValue("@xmlParameter", newXml);

        int result = command.ExecuteNonQuery();
        Console.WriteLine(result + " row was added.");
        Console.WriteLine("Press Enter to continue.");
        Console.ReadLine();
    }
  }

    private static string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Integrated Security=true;" +
        "Initial Catalog=AdventureWorks; ";
    }
}
```

See also

- SqlXml
- XML Data in SQL Server
- ADO.NET Overview

SQL Server Binary and Large-Value Data

4/26/2022 • 2 minutes to read • Edit Online

SQL Server provides the `max` specifier, which expands the storage capacity of the `varchar`, `nvarchar`, and `varbinary` data types. `varchar(max)`, `nvarchar(max)`, and `varbinary(max)` are collectively called *large-value data types*. You can use the large-value data types to store up to 2^31-1 bytes of data.

SQL Server 2008 introduces the FILESTREAM attribute, which is not a data type, but rather an attribute that can be defined on a column, allowing large-value data to be stored on the file system instead of in the database.

In This Section

Modifying Large-Value (max) Data in ADO.NET
Describes how to work with the large-value data types.

FILESTREAM Data
Describes how to work with large-value data stored in SQL Server 2008 with the FILESTREAM attribute.

See also

- SQL Server Data Types and ADO.NET
- SQL Server Data Operations in ADO.NET
- Retrieving and Modifying Data in ADO.NET
- ADO.NET Overview

Modifying Large-Value (max) Data in ADO.NET

4/26/2022 • 10 minutes to read • Edit Online

Large object (LOB) data types are those that exceed the maximum row size of 8 kilobytes (KB). SQL Server provides a `max` specifier for `varchar`, `nvarchar`, and `varbinary` data types to allow storage of values as large as 2^32 bytes. Table columns and Transact-SQL variables may specify `varchar(max)`, `nvarchar(max)`, or `varbinary(max)` data types. In ADO.NET, the `max` data types can be fetched by a `DataReader`, and can also be specified as both input and output parameter values without any special handling. For large `varchar` data types, data can be retrieved and updated incrementally.

The `max` data types can be used for comparisons, as Transact-SQL variables, and for concatenation. They can also be used in the DISTINCT, ORDER BY, GROUP BY clauses of a SELECT statement as well as in aggregates, joins, and subqueries.

For more information, see Using Large-Value Data Types.

Large-Value Type Restrictions

The following restrictions apply to the `max` data types, which do not exist for smaller data types:

- A `sql_variant` cannot contain a large `varchar` data type.
- Large `varchar` columns cannot be specified as a key column in an index. They are allowed in an included column in a non-clustered index.
- Large `varchar` columns cannot be used as partitioning key columns.

Working with Large-Value Types in Transact-SQL

The Transact-SQL `OPENROWSET` function is a one-time method of connecting and accessing remote data. It includes all of the connection information necessary to access remote data from an OLE DB data source. `OPENROWSET` can be referenced in the FROM clause of a query as though it were a table name. It can also be referenced as the target table of an INSERT, UPDATE, or DELETE statement, subject to the capabilities of the OLE DB provider.

The `OPENROWSET` function includes the `BULK` rowset provider, which allows you to read data directly from a file without loading the data into a target table. This enables you to use `OPENROWSET` in a simple INSERT SELECT statement.

The `OPENROWSET BULK` option arguments provide significant control over where to begin and end reading data, how to deal with errors, and how data is interpreted. For example, you can specify that the data file be read as a single-row, single-column rowset of type `varbinary`, `varchar`, or `nvarchar`.

The following example inserts a photo into the ProductPhoto table in the AdventureWorks sample database. When using the `BULK OPENROWSET` provider, you must supply the named list of columns even if you aren't inserting values into every column. The primary key in this case is defined as an identity column, and may be omitted from the column list. Note that you must also supply a correlation name at the end of the `OPENROWSET` statement, which in this case is ThumbnailPhoto. This correlates with the column in the `ProductPhoto` table into which the file is being loaded.

```
INSERT Production.ProductPhoto (
    ThumbnailPhoto,
    ThumbnailPhotoFilePath,
    LargePhoto,
    LargePhotoFilePath)
SELECT ThumbnailPhoto.*, null, null, N'tricycle_pink.gif'
FROM OPENROWSET
    (BULK 'c:\images\tricycle.jpg', SINGLE_BLOB) ThumbnailPhoto
```

Updating Data Using UPDATE .WRITE

The Transact-SQL UPDATE statement has new WRITE syntax for modifying the contents of `varchar(max)`, `nvarchar(max)`, or `varbinary(max)` columns. This allows you to perform partial updates of the data. The UPDATE .WRITE syntax is shown here in abbreviated form:

UPDATE

{ *<object>* }

SET

{ *column_name* = { .WRITE (*expression* , @Offset , @Length) }

The WRITE method specifies that a section of the value of the *column_name* will be modified. The expression is the value that will be copied to the *column_name*, the `@Offset` is the beginning point at which the expression will be written, and the `@Length` argument is the length of the section in the column.

IF	THEN
The expression is set to NULL	`@Length` is ignored and the value in *column_name* is truncated at the specified `@Offset`.
`@Offset` is NULL	The update operation appends the expression at the end of the existing *column_name* value and `@Length` is ignored.
`@Offset` is greater than the length of the column_name value	SQL Server returns an error.
`@Length` is NULL	The update operation removes all data from `@Offset` to the end of the `column_name` value.

> **NOTE**
> Neither `@Offset` nor `@Length` can be a negative number.

Example

This Transact-SQL example updates a partial value in DocumentSummary, an `nvarchar(max)` column in the Document table in the AdventureWorks database. The word 'components' is replaced by the word 'features' by specifying the replacement word, the beginning location (offset) of the word to be replaced in the existing data, and the number of characters to be replaced (length). The example includes SELECT statements before and after the UPDATE statement to compare results.

```
USE AdventureWorks;
GO
--View the existing value.
SELECT DocumentSummary
FROM Production.Document
WHERE DocumentID = 3;
GO
-- The first sentence of the results will be:
-- Reflectors are vital safety components of your bicycle.

--Modify a single word in the DocumentSummary column
UPDATE Production.Document
SET DocumentSummary .WRITE (N'features',28,10)
WHERE DocumentID = 3 ;
GO
--View the modified value.
SELECT DocumentSummary
FROM Production.Document
WHERE DocumentID = 3;
GO
-- The first sentence of the results will be:
-- Reflectors are vital safety features of your bicycle.
```

Working with Large-Value Types in ADO.NET

You can work with large value types in ADO.NET by specifying large value types as SqlParameter objects in a SqlDataReader to return a result set, or by using a SqlDataAdapter to fill a `DataSet` / `DataTable`. There is no difference between the way you work with a large value type and its related, smaller value data type.

Using GetSqlBytes to Retrieve Data

The `GetSqlBytes` method of the SqlDataReader can be used to retrieve the contents of a `varbinary(max)` column. The following code fragment assumes a SqlCommand object named `cmd` that selects `varbinary(max)` data from a table and a SqlDataReader object named `reader` that retrieves the data as SqlBytes.

```
reader = cmd.ExecuteReader(CommandBehavior.CloseConnection)
While reader.Read()
    Dim bytes As SqlBytes = reader.GetSqlBytes(0)
End While
```

```
reader = cmd.ExecuteReader(CommandBehavior.CloseConnection);
while (reader.Read())
    {
        SqlBytes bytes = reader.GetSqlBytes(0);
    }
```

Using GetSqlChars to Retrieve Data

The `GetSqlChars` method of the SqlDataReader can be used to retrieve the contents of a `varchar(max)` or `nvarchar(max)` column. The following code fragment assumes a SqlCommand object named `cmd` that selects `nvarchar(max)` data from a table and a SqlDataReader object named `reader` that retrieves the data.

```
reader = cmd.ExecuteReader(CommandBehavior.CloseConnection)
While reader.Read()
    Dim buffer As SqlChars = reader.GetSqlChars(0)
End While
```

```
reader = cmd.ExecuteReader(CommandBehavior.CloseConnection);
while (reader.Read())
{
    SqlChars buffer = reader.GetSqlChars(0);
}
```

Using GetSqlBinary to Retrieve Data

The `GetSqlBinary` method of a SqlDataReader can be used to retrieve the contents of a `varbinary(max)` column. The following code fragment assumes a SqlCommand object named `cmd` that selects `varbinary(max)` data from a table and a SqlDataReader object named `reader` that retrieves the data as a SqlBinary stream.

```
reader = cmd.ExecuteReader(CommandBehavior.CloseConnection)
While reader.Read()
    Dim binaryStream As SqlBinary = reader.GetSqlBinary(0)
End While
```

```
reader = cmd.ExecuteReader(CommandBehavior.CloseConnection);
while (reader.Read())
    {
        SqlBinary binaryStream = reader.GetSqlBinary(0);
    }
```

Using GetBytes to Retrieve Data

The `GetBytes` method of a SqlDataReader reads a stream of bytes from the specified column offset into a byte array starting at the specified array offset. The following code fragment assumes a SqlDataReader object named `reader` that retrieves bytes into a byte array. Note that, unlike `GetSqlBytes`, `GetBytes` requires a size for the array buffer.

```
While reader.Read()
    Dim buffer(4000) As Byte
    Dim byteCount As Integer = _
    CInt(reader.GetBytes(1, 0, buffer, 0, 4000))
End While
```

```
while (reader.Read())
{
    byte[] buffer = new byte[4000];
    long byteCount = reader.GetBytes(1, 0, buffer, 0, 4000);
}
```

Using GetValue to Retrieve Data

The `GetValue` method of a SqlDataReader reads the value from the specified column offset into an array. The following code fragment assumes a SqlDataReader object named `reader` that retrieves binary data from the first column offset, and then string data from the second column offset.

```
While reader.Read()
    ' Read the data from varbinary(max) column
    Dim binaryData() As Byte = CByte(reader.GetValue(0))

    ' Read the data from varchar(max) or nvarchar(max) column
    Dim stringData() As String = Cstr((reader.GetValue(1))
End While
```

```
while (reader.Read())
{
    // Read the data from varbinary(max) column
    byte[] binaryData = (byte[])reader.GetValue(0);

    // Read the data from varchar(max) or nvarchar(max) column
    String stringData = (String)reader.GetValue(1);
}
```

Converting from Large Value Types to CLR Types

You can convert the contents of a `varchar(max)` or `nvarchar(max)` column using any of the string conversion methods, such as `ToString`. The following code fragment assumes a SqlDataReader object named `reader` that retrieves the data.

```
While reader.Read()
    Dim str as String = reader(0).ToString()
    Console.WriteLine(str)
End While
```

```
while (reader.Read())
{
     string str = reader[0].ToString();
     Console.WriteLine(str);
}
```

Example

The following code retrieves the name and the `LargePhoto` object from the `ProductPhoto` table in the `AdventureWorks` database and saves it to a file. The assembly needs to be compiled with a reference to the System.Drawing namespace. The GetSqlBytes method of the SqlDataReader returns a SqlBytes object that exposes a `Stream` property. The code uses this to create a new `Bitmap` object, and then saves it in the Gif `ImageFormat`.

```
static private void TestGetSqlBytes(int documentID, string filePath)
{
    // Assumes GetConnectionString returns a valid connection string.
    using (SqlConnection connection =
               new SqlConnection(GetConnectionString()))
    {
        SqlCommand command = connection.CreateCommand();
        SqlDataReader reader = null;
        try
        {
            // Setup the command
            command.CommandText =
                "SELECT LargePhotoFileName, LargePhoto "
                + "FROM Production.ProductPhoto "
                + "WHERE ProductPhotoID=@ProductPhotoID";
            command.CommandType = CommandType.Text;

            // Declare the parameter
            SqlParameter paramID =
                new SqlParameter("@ProductPhotoID", SqlDbType.Int);
            paramID.Value = documentID;
            command.Parameters.Add(paramID);
            connection.Open();

            string photoName = null;
```

```
            reader = command.ExecuteReader(CommandBehavior.CloseConnection);

            if (reader.HasRows)
            {
                while (reader.Read())
                {
                    // Get the name of the file.
                    photoName = reader.GetString(0);

                    // Ensure that the column isn't null
                    if (reader.IsDBNull(1))
                    {
                        Console.WriteLine("{0} is unavailable.", photoName);
                    }
                    else
                    {
                        SqlBytes bytes = reader.GetSqlBytes(1);
                        using (Bitmap productImage = new Bitmap(bytes.Stream))
                        {
                            String fileName = filePath + photoName;

                            // Save in gif format.
                            productImage.Save(fileName, ImageFormat.Gif);
                            Console.WriteLine("Successfully created {0}.", fileName);
                        }
                    }
                }
            }
            else
            {
                Console.WriteLine("No records returned.");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            if (reader != null)
                reader.Dispose();
        }
    }
}
```

```
Private Sub GetPhoto( _
  ByVal documentID As Integer, ByVal filePath As String)
    ' Assumes GetConnectionString returns a valid connection string.
    Using connection As New SqlConnection(GetConnectionString())
        Dim command As SqlCommand = connection.CreateCommand()
        Dim reader As SqlDataReader
        Try
            ' Setup the command
            command.CommandText = _
              "SELECT LargePhotoFileName, LargePhoto FROM" _
                & " Production.ProductPhoto" _
                & " WHERE ProductPhotoID=@ProductPhotoID"
            command.CommandType = CommandType.Text

            ' Declare the parameter
            Dim paramID As SqlParameter = _
                New SqlParameter("@ProductPhotoID", SqlDbType.Int)
            paramID.Value = documentID
            command.Parameters.Add(paramID)
            connection.Open()

            Dim photoName As String

            reader = _
             command.ExecuteReader(CommandBehavior.CloseConnection)

            If reader.HasRows Then
                While reader.Read()
                    ' Get the name of the file
                    photoName = reader.GetString(0)

                    ' Ensure that the column isn't null
                    If (reader.IsDBNull(1)) Then
                        Console.WriteLine("{0} is unavailable.", photoName)
                    Else
                        Dim bytes As SqlBytes = reader.GetSqlBytes(1)
                        Using productImage As Bitmap = _
                          New Bitmap(bytes.Stream)
                            Dim fileName As String = filePath & photoName

                            ' Save in gif format.
                            productImage.Save( _
                              fileName, ImageFormat.Gif)
                            Console.WriteLine("Successfully created {0}.", fileName)
                        End Using
                    End If
                End While
            Else
                Console.WriteLine("No records returned.")
            End If
        Catch ex As Exception
            Console.WriteLine("Exception: {0}", ex.Message)
        End Try
    End Using
End Sub
```

Using Large Value Type Parameters

Large value types can be used in SqlParameter objects the same way you use smaller value types in SqlParameter objects. You can retrieve large value types as SqlParameter values, as shown in the following example. The code assumes that the following GetDocumentSummary stored procedure exists in the AdventureWorks sample database. The stored procedure takes an input parameter named @DocumentID and returns the contents of the DocumentSummary column in the @DocumentSummary output parameter.

```
CREATE PROCEDURE GetDocumentSummary
(
    @DocumentID int,
    @DocumentSummary nvarchar(MAX) OUTPUT
)
AS
SET NOCOUNT ON
SELECT  @DocumentSummary=Convert(nvarchar(MAX), DocumentSummary)
FROM    Production.Document
WHERE   DocumentID=@DocumentID
```

Example

The ADO.NET code creates SqlConnection and SqlCommand objects to execute the GetDocumentSummary stored procedure and retrieve the document summary, which is stored as a large value type. The code passes a value for the @DocumentID input parameter, and displays the results passed back in the @DocumentSummary output parameter in the Console window.

```
static private string GetDocumentSummary(int documentID)
{
    //Assumes GetConnectionString returns a valid connection string.
    using (SqlConnection connection =
               new SqlConnection(GetConnectionString()))
    {
        connection.Open();
        SqlCommand command = connection.CreateCommand();
        try
        {
            // Setup the command to execute the stored procedure.
            command.CommandText = "GetDocumentSummary";
            command.CommandType = CommandType.StoredProcedure;

            // Set up the input parameter for the DocumentID.
            SqlParameter paramID =
                new SqlParameter("@DocumentID", SqlDbType.Int);
            paramID.Value = documentID;
            command.Parameters.Add(paramID);

            // Set up the output parameter to retrieve the summary.
            SqlParameter paramSummary =
                new SqlParameter("@DocumentSummary",
                SqlDbType.NVarChar, -1);
            paramSummary.Direction = ParameterDirection.Output;
            command.Parameters.Add(paramSummary);

            // Execute the stored procedure.
            command.ExecuteNonQuery();
            Console.WriteLine((String)(paramSummary.Value));
            return (String)(paramSummary.Value);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
            return null;
        }
    }
}
```

```
Private Function GetDocumentSummary( _
  ByVal documentID As Integer) As String

    ' Assumes GetConnectionString returns a valid connection string.
    Using connection As New SqlConnection(GetConnectionString())
        connection.Open()
        Dim command As SqlCommand = connection.CreateCommand()

        ' Setup the command to execute the stored procedure.
        command.CommandText = "GetDocumentSummary"
        command.CommandType = CommandType.StoredProcedure

        ' Set up the input parameter for the DocumentID.
        Dim paramID As SqlParameter = _
            New SqlParameter("@DocumentID", SqlDbType.Int)
        paramID.Value = documentID
        command.Parameters.Add(paramID)

        ' Set up the output parameter to retrieve the summary.
        Dim paramSummary As SqlParameter = _
            New SqlParameter("@DocumentSummary", _
               SqlDbType.NVarChar, -1)
        paramSummary.Direction = ParameterDirection.Output
        command.Parameters.Add(paramSummary)

        ' Execute the stored procedure.
        command.ExecuteNonQuery()
        Console.WriteLine(paramSummary.Value)
        Return paramSummary.Value.ToString
    End Using
End Function
```

See also

- SQL Server Binary and Large-Value Data
- SQL Server Data Type Mappings
- SQL Server Data Operations in ADO.NET
- ADO.NET Overview

FILESTREAM Data

4/26/2022 • 4 minutes to read • Edit Online

The FILESTREAM storage attribute is for binary (BLOB) data stored in a varbinary(max) column. Before FILESTREAM, storing binary data required special handling. Unstructured data, such as text documents, images and video, is often stored outside of the database, making it difficult to manage.

> **NOTE**
>
> You must install the .NET Framework 3.5 SP1 (or later) to work with FILESTREAM data using SqlClient.

Specifying the FILESTREAM attribute on a varbinary(max) column causes SQL Server to store the data on the local NTFS file system instead of in the database file. Although it is stored separately, you can use the same Transact-SQL statements that are supported for working with varbinary(max) data that is stored in the database.

SqlClient Support for FILESTREAM

The .NET Framework Data Provider for SQL Server, System.Data.SqlClient, supports reading and writing to FILESTREAM data using the SqlFileStream class defined in the System.Data.SqlTypes namespace. `SqlFileStream` inherits from the Stream class, which provides methods for reading and writing to streams of data. Reading from a stream transfers data from the stream into a data structure, such as an array of bytes. Writing transfers the data from the data structure into a stream.

Creating the SQL Server Table

The following Transact-SQL statements creates a table named employees and inserts a row of data. Once you have enabled FILESTREAM storage, you can use this table in conjunction with the code examples that follow.

```
CREATE TABLE employees
(
  EmployeeId INT  NOT NULL  PRIMARY KEY,
  Photo VARBINARY(MAX) FILESTREAM  NULL,
  RowGuid UNIQUEIDENTIFIER  NOT NULL  ROWGUIDCOL
  UNIQUE DEFAULT NEWID()
)
GO
Insert into employees
Values(1, 0x00, default)
GO
```

Example: Reading, Overwriting, and Inserting FILESTREAM Data

The following sample demonstrates how to read data from a FILESTREAM. The code gets the logical path to the file, setting the `FileAccess` to `Read` and the `FileOptions` to `SequentialScan`. The code then reads the bytes from the SqlFileStream into the buffer. The bytes are then written to the console window.

The sample also demonstrates how to write data to a FILESTREAM in which all existing data is overwritten. The code gets the logical path to the file and creates the `SqlFileStream`, setting the `FileAccess` to `Write` and the `FileOptions` to `SequentialScan`. A single byte is written to the `SqlFileStream`, replacing any data in the file.

The sample also demonstrates how to write data to a FILESTREAM by using the Seek method to append data to the end of the file. The code gets the logical path to the file and creates the `SqlFileStream`, setting the `FileAccess` to `ReadWrite` and the `FileOptions` to `SequentialScan`. The code uses the Seek method to seek to

the end of the file, appending a single byte to the existing file.

```
using System;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Data;
using System.IO;

namespace FileStreamTest
{
    class Program
    {
        static void Main(string[] args)
        {
            SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder("server=(local);integrated
security=true;database=myDB");
            ReadFileStream(builder);
            OverwriteFileStream(builder);
            InsertFileStream(builder);

            Console.WriteLine("Done");
        }

        private static void ReadFileStream(SqlConnectionStringBuilder connStringBuilder)
        {
            using (SqlConnection connection = new SqlConnection(connStringBuilder.ToString()))
            {
                connection.Open();
                SqlCommand command = new SqlCommand("SELECT TOP(1) Photo.PathName(),
GET_FILESTREAM_TRANSACTION_CONTEXT() FROM employees", connection);

                SqlTransaction tran = connection.BeginTransaction(IsolationLevel.ReadCommitted);
                command.Transaction = tran;

                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        // Get the pointer for the file
                        string path = reader.GetString(0);
                        byte[] transactionContext = reader.GetSqlBytes(1).Buffer;

                        // Create the SqlFileStream
                        using (Stream fileStream = new SqlFileStream(path, transactionContext,
FileAccess.Read, FileOptions.SequentialScan, allocationSize: 0))
                        {
                            // Read the contents as bytes and write them to the console
                            for (long index = 0; index < fileStream.Length; index++)
                            {
                                Console.WriteLine(fileStream.ReadByte());
                            }
                        }
                    }
                }
                tran.Commit();
            }
        }

        private static void OverwriteFileStream(SqlConnectionStringBuilder connStringBuilder)
        {
            using (SqlConnection connection = new SqlConnection(connStringBuilder.ToString()))
            {
                connection.Open();

                SqlCommand command = new SqlCommand("SELECT TOP(1) Photo.PathName(),
GET_FILESTREAM_TRANSACTION_CONTEXT() FROM employees", connection);

                SqlTransaction tran = connection.BeginTransaction(IsolationLevel.ReadCommitted);
```

```
                command.Transaction = tran;

                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        // Get the pointer for file
                        string path = reader.GetString(0);
                        byte[] transactionContext = reader.GetSqlBytes(1).Buffer;

                        // Create the SqlFileStream
                        using (Stream fileStream = new SqlFileStream(path, transactionContext,
FileAccess.Write, FileOptions.SequentialScan, allocationSize: 0))
                        {
                            // Write a single byte to the file. This will
                            // replace any data in the file.
                            fileStream.WriteByte(0x01);
                        }
                    }
                }
                tran.Commit();
            }
        }

        private static void InsertFileStream(SqlConnectionStringBuilder connStringBuilder)
        {
            using (SqlConnection connection = new SqlConnection(connStringBuilder.ToString()))
            {
                connection.Open();

                SqlCommand command = new SqlCommand("SELECT TOP(1) Photo.PathName(),
GET_FILESTREAM_TRANSACTION_CONTEXT() FROM employees", connection);

                SqlTransaction tran = connection.BeginTransaction(IsolationLevel.ReadCommitted);
                command.Transaction = tran;

                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        // Get the pointer for file
                        string path = reader.GetString(0);
                        byte[] transactionContext = reader.GetSqlBytes(1).Buffer;

                        using (Stream fileStream = new SqlFileStream(path, transactionContext,
FileAccess.ReadWrite, FileOptions.SequentialScan, allocationSize: 0))
                        {
                            // Seek to the end of the file
                            fileStream.Seek(0, SeekOrigin.End);

                            // Append a single byte
                            fileStream.WriteByte(0x01);
                        }
                    }
                }
                tran.Commit();
            }

        }
    }
}
```

For another sample, see How to store and fetch binary data into a file stream column.

SQL Server docs resources

The complete documentation for FILESTREAM is located in the following sections of the SQL Server docs.

TOPIC	DESCRIPTION
FILESTREAM (SQL Server)	Describes when to use FILESTREAM storage and how it integrates the SQL Server Database Engine with an NTFS file system.
Create Client Applications for FILESTREAM Data	Describes the Windows API functions for working with FILESTREAM data.
FILESTREAM and Other SQL Server Features	Provides considerations, guidelines and limitations for using FILESTREAM data with other features of SQL Server.

See also

- SQL Server Data Types and ADO.NET
- Retrieving and Modifying Data in ADO.NET
- Code Access Security and ADO.NET
- SQL Server Binary and Large-Value Data
- ADO.NET Overview

Inserting an Image from a File

4/26/2022 • 2 minutes to read • Edit Online

You can write a binary large object (BLOB) to a database as either binary or character data, depending on the type of field at your data source. BLOB is a generic term that refers to the `text`, `ntext`, and `image` data types, which typically contain documents and pictures.

To write a BLOB value to your database, issue the appropriate INSERT or UPDATE statement and pass the BLOB value as an input parameter (see Configuring Parameters and Parameter Data Types). If your BLOB is stored as text, such as a SQL Server `text` field, you can pass the BLOB as a string parameter. If the BLOB is stored in binary format, such as a SQL Server `image` field, you can pass an array of type `byte` as a binary parameter.

Example

The following code example adds employee information to the Employees table in the Northwind database. A photo of the employee is read from a file and added to the Photo field in the table, which is an image field.

```
Public Shared Sub AddEmployee( _
  lastName As String, _
  firstName As String, _
  title As String, _
  hireDate As DateTime, _
  reportsTo As Integer, _
  photoFilePath As String, _
  connectionString As String)

  Dim photo() as Byte = GetPhoto(photoFilePath)

  Using connection As SqlConnection = New SqlConnection( _
    connectionString)

  Dim command As SqlCommand = New SqlCommand( _
    "INSERT INTO Employees (LastName, FirstName, Title, " & _
    "HireDate, ReportsTo, Photo) " & _
    "Values(@LastName, @FirstName, @Title, " & _
    "@HireDate, @ReportsTo, @Photo)", connection)

  command.Parameters.Add("@LastName",  _
    SqlDbType.NVarChar, 20).Value = lastName
  command.Parameters.Add("@FirstName", _
    SqlDbType.NVarChar, 10).Value = firstName
  command.Parameters.Add("@Title", _
    SqlDbType.NVarChar, 30).Value = title
  command.Parameters.Add("@HireDate", _
    SqlDbType.DateTime).Value = hireDate
  command.Parameters.Add("@ReportsTo", _
    SqlDbType.Int).Value = reportsTo

  command.Parameters.Add("@Photo", _
    SqlDbType.Image, photo.Length).Value = photo

  connection.Open()
  command.ExecuteNonQuery()

  End Using
End Sub

Public Shared Function GetPhoto(filePath As String) As Byte()
  Dim stream As FileStream = new FileStream( _
     filePath, FileMode.Open, FileAccess.Read)
  Dim reader As BinaryReader = new BinaryReader(stream)

  Dim photo() As Byte = reader.ReadBytes(stream.Length)

  reader.Close()
  stream.Close()

  Return photo
End Function
```

```
public static void AddEmployee(
  string lastName,
  string firstName,
  string title,
  DateTime hireDate,
  int reportsTo,
  string photoFilePath,
  string connectionString)
{
  byte[] photo = GetPhoto(photoFilePath);

  using (SqlConnection connection = new SqlConnection(
    connectionString))

  SqlCommand command = new SqlCommand(
    "INSERT INTO Employees (LastName, FirstName, " +
    "Title, HireDate, ReportsTo, Photo) " +
    "Values(@LastName, @FirstName, @Title, " +
    "@HireDate, @ReportsTo, @Photo)", connection);

  command.Parameters.Add("@LastName",
     SqlDbType.NVarChar, 20).Value = lastName;
  command.Parameters.Add("@FirstName",
      SqlDbType.NVarChar, 10).Value = firstName;
  command.Parameters.Add("@Title",
      SqlDbType.NVarChar, 30).Value = title;
  command.Parameters.Add("@HireDate",
       SqlDbType.DateTime).Value = hireDate;
  command.Parameters.Add("@ReportsTo",
      SqlDbType.Int).Value = reportsTo;

  command.Parameters.Add("@Photo",
      SqlDbType.Image, photo.Length).Value = photo;

  connection.Open();
  command.ExecuteNonQuery();
  }
}

public static byte[] GetPhoto(string filePath)
{
  FileStream stream = new FileStream(
      filePath, FileMode.Open, FileAccess.Read);
  BinaryReader reader = new BinaryReader(stream);

  byte[] photo = reader.ReadBytes((int)stream.Length);

  reader.Close();
  stream.Close();

  return photo;
}
```

See also

- Using Commands to Modify Data
- Retrieving Binary Data
- SQL Server Binary and Large-Value Data
- SQL Server Data Type Mappings
- ADO.NET Overview

SQL Server Data Operations in ADO.NET

4/26/2022 • 2 minutes to read • Edit Online

This section describes SQL Server features and functionality that are specific to the .NET Framework Data Provider for SQL Server (System.Data.SqlClient).

In This Section

Bulk Copy Operations in SQL Server
Describes the bulk copy functionality for the .NET Data Provider for SQL Server.

Multiple Active Result Sets (MARS)
Describes how to have more than one SqlDataReader open on a connection when each instance of SqlDataReader is started from a separate command.

Asynchronous Operations
Describes how to perform asynchronous database operations by using an API that is modeled after the asynchronous model used by the .NET Framework.

Table-Valued Parameters
Describes how to work with table-valued parameters, which were introduced in SQL Server 2008.

See also

- Retrieving and Modifying Data in ADO.NET
- SQL Server and ADO.NET
- ADO.NET Overview

Bulk Copy Operations in SQL Server

4/26/2022 • 2 minutes to read • Edit Online

Microsoft SQL Server includes a popular command-line utility named **bcp** for quickly bulk copying large files into tables or views in SQL Server databases. The SqlBulkCopy class allows you to write managed code solutions that provide similar functionality. There are other ways to load data into a SQL Server table (INSERT statements, for example) but SqlBulkCopy offers a significant performance advantage over them.

The SqlBulkCopy class can be used to write data only to SQL Server tables. But the data source is not limited to SQL Server; any data source can be used, as long as the data can be loaded to a DataTable instance or read with a IDataReader instance.

Using the SqlBulkCopy class, you can perform:

- A single bulk copy operation
- Multiple bulk copy operations
- A bulk copy operation within a transaction

> **NOTE**
> When using .NET Framework version 1.1 or earlier (which does not support the SqlBulkCopy class), you can execute the SQL Server Transact-SQL **BULK INSERT** statement using the SqlCommand object.

In This Section

Bulk Copy Example Setup
Describes the tables used in the bulk copy examples and provides SQL scripts for creating the tables in the AdventureWorks database.

Single Bulk Copy Operations
Describes how to do a single bulk copy of data into an instance of SQL Server using the SqlBulkCopy class, and how to perform the bulk copy operation using Transact-SQL statements and the SqlCommand class.

Multiple Bulk Copy Operations
Describes how to do multiple bulk copy operations of data into an instance of SQL Server using the SqlBulkCopy class.

Transaction and Bulk Copy Operations
Describes how to perform a bulk copy operation within a transaction, including how to commit or rollback the transaction.

See also

- SQL Server and ADO.NET
- ADO.NET Overview

Bulk Copy Example Setup

4/26/2022 • 2 minutes to read • Edit Online

The SqlBulkCopy class can be used to write data only to SQL Server tables. The code samples shown in this topic use the SQL Server sample database, **AdventureWorks**. To avoid altering the existing tables code samples write data to tables that you must create first.

The **BulkCopyDemoMatchingColumns** and **BulkCopyDemoDifferentColumns** tables are both based on the **AdventureWorks Production.Products** table. In code samples that use these tables, data is added from the **Production.Products** table to one of these sample tables. The **BulkCopyDemoDifferentColumns** table is used when the sample illustrates how to map columns from the source data to the destination table; **BulkCopyDemoMatchingColumns** is used for most other samples.

A few of the code samples demonstrate how to use one SqlBulkCopy class to write to multiple tables. For these samples, the **BulkCopyDemoOrderHeader** and **BulkCopyDemoOrderDetail** tables are used as the destination tables. These tables are based on the **Sales.SalesOrderHeader** and **Sales.SalesOrderDetail** tables in **AdventureWorks**.

> **NOTE**
>
> The **SqlBulkCopy** code samples are provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT … SELECT` statement to copy the data.

Table Setup

To create the tables necessary for the code samples to run correctly, you must run the following Transact-SQL statements in a SQL Server database.

```
USE AdventureWorks

IF EXISTS (SELECT * FROM dbo.sysobjects
 WHERE id = object_id(N'[dbo].[BulkCopyDemoMatchingColumns]')
 AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
    DROP TABLE [dbo].[BulkCopyDemoMatchingColumns]

CREATE TABLE [dbo].[BulkCopyDemoMatchingColumns]([ProductID] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](50) NOT NULL,
    [ProductNumber] [nvarchar](25) NOT NULL,
 CONSTRAINT [PK_ProductID] PRIMARY KEY CLUSTERED
(
    [ProductID] ASC
) ON [PRIMARY]) ON [PRIMARY]

IF EXISTS (SELECT * FROM dbo.sysobjects
 WHERE id = object_id(N'[dbo].[BulkCopyDemoDifferentColumns]')
 AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
    DROP TABLE [dbo].[BulkCopyDemoDifferentColumns]

CREATE TABLE [dbo].[BulkCopyDemoDifferentColumns]([ProdID] [int] IDENTITY(1,1) NOT NULL,
    [ProdNum] [nvarchar](25) NOT NULL,
    [ProdName] [nvarchar](50) NOT NULL,
 CONSTRAINT [PK_ProdID] PRIMARY KEY CLUSTERED
(
    [ProdID] ASC
) ON [PRIMARY]) ON [PRIMARY]

IF EXISTS (SELECT * FROM dbo.sysobjects
 WHERE id = object_id(N'[dbo].[BulkCopyDemoOrderHeader]')
 AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
    DROP TABLE [dbo].[BulkCopyDemoOrderHeader]

CREATE TABLE [dbo].[BulkCopyDemoOrderHeader]([SalesOrderID] [int] IDENTITY(1,1) NOT NULL,
    [OrderDate] [datetime] NOT NULL,
    [AccountNumber] [nvarchar](15) NULL,
 CONSTRAINT [PK_SalesOrderID] PRIMARY KEY CLUSTERED
(
    [SalesOrderID] ASC
) ON [PRIMARY]) ON [PRIMARY]

IF EXISTS (SELECT * FROM dbo.sysobjects
 WHERE id = object_id(N'[dbo].[BulkCopyDemoOrderDetail]')
 AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
    DROP TABLE [dbo].[BulkCopyDemoOrderDetail]

CREATE TABLE [dbo].[BulkCopyDemoOrderDetail]([SalesOrderID] [int] NOT NULL,
    [SalesOrderDetailID] [int] NOT NULL,
    [OrderQty] [smallint] NOT NULL,
    [ProductID] [int] NOT NULL,
    [UnitPrice] [money] NOT NULL,
 CONSTRAINT [PK_LineNumber] PRIMARY KEY CLUSTERED
(
    [SalesOrderID] ASC,
    [SalesOrderDetailID] ASC
) ON [PRIMARY]) ON [PRIMARY]
```

See also

- Bulk Copy Operations in SQL Server
- ADO.NET Overview

Single Bulk Copy Operations

4/26/2022 • 5 minutes to read • Edit Online

The simplest approach to performing a SQL Server bulk copy operation is to perform a single operation against a database. By default, a bulk copy operation is performed as an isolated operation: the copy operation occurs in a non-transacted way, with no opportunity for rolling it back.

> **NOTE**
>
> If you need to roll back all or part of the bulk copy when an error occurs, you can either use a SqlBulkCopy-managed transaction, or perform the bulk copy operation within an existing transaction. **SqlBulkCopy** will also work with System.Transactions if the connection is enlisted (implicitly or explicitly) into a **System.Transactions** transaction.
>
> For more information, see Transaction and Bulk Copy Operations.

The general steps for performing a bulk copy operation are as follows:

1. Connect to the source server and obtain the data to be copied. Data can also come from other sources, if it can be retrieved from an IDataReader or DataTable object.
2. Connect to the destination server (unless you want **SqlBulkCopy** to establish a connection for you).
3. Create a SqlBulkCopy object, setting any necessary properties.
4. Set the **DestinationTableName** property to indicate the target table for the bulk insert operation.
5. Call one of the **WriteToServer** methods.
6. Optionally, update properties and call **WriteToServer** again as necessary.
7. Call Close, or wrap the bulk copy operations within a `Using` statement.

Caution

We recommend that the source and target column data types match. If the data types do not match, **SqlBulkCopy** attempts to convert each source value to the target data type, using the rules employed by Value. Conversions can affect performance, and also can result in unexpected errors. For example, a `Double` data type can be converted to a `Decimal` data type most of the time, but not always.

Example

The following console application demonstrates how to load data using the SqlBulkCopy class. In this example, a SqlDataReader is used to copy data from the **Production.Product** table in the SQL Server **AdventureWorks** database to a similar table in the same database.

> **IMPORTANT**
>
> This sample will not run unless you have created the work tables as described in Bulk Copy Example Setup. This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT … SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
```

```
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
                   new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoMatchingColumns;",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Open the destination connection. In the real world you would
            // not use SqlBulkCopy to move data from one table to the other
            // in the same database. This is for demonstration purposes only.
            using (SqlConnection destinationConnection =
                       new SqlConnection(connectionString))
            {
                destinationConnection.Open();

                // Set up the bulk copy object.
                // Note that the column positions in the source
                // data reader match the column positions in
                // the destination table so there is no need to
                // map columns.
                using (SqlBulkCopy bulkCopy =
                           new SqlBulkCopy(destinationConnection))
                {
                    bulkCopy.DestinationTableName =
                        "dbo.BulkCopyDemoMatchingColumns";

                    try
                    {
                        // Write from the source to the destination.
                        bulkCopy.WriteToServer(reader);
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(ex.Message);
                    }
                    finally
                    {
                        // Close the SqlDataReader. The SqlBulkCopy
                        // object is automatically closed at the end
                        // of the using block.
                        reader.Close();
                    }
                }

                // Perform a final count on the destination
                // table to see how many rows were added.
                long countEnd = System.Convert.ToInt32(
                    commandRowCount.ExecuteScalar());
                Console.WriteLine("Ending row count = {0}", countEnd);
```

```
                Console.WriteLine("{0} rows were added.", countEnd - countStart);
                Console.WriteLine("Press Enter to finish.");
                Console.ReadLine();
            }
        }
    }

    private static string GetConnectionString()
        // To avoid storing the sourceConnection string in your code,
        // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}
```

```
Imports System.Data.SqlClient

Module Module1
    Sub Main()
        Dim connectionString As String = GetConnectionString()

        ' Open a connection to the AdventureWorks database.
        Using sourceConnection As SqlConnection = _
           New SqlConnection(connectionString)
            sourceConnection.Open()

            ' Perform an initial count on the destination table.
            Dim commandRowCount As New SqlCommand( _
            "SELECT COUNT(*) FROM dbo.BulkCopyDemoMatchingColumns;", _
                sourceConnection)
            Dim countStart As Long = _
               System.Convert.ToInt32(commandRowCount.ExecuteScalar())
            Console.WriteLine("Starting row count = {0}", countStart)

            ' Get data from the source table as a SqlDataReader.
            Dim commandSourceData As New SqlCommand( _
               "SELECT ProductID, Name, ProductNumber " & _
               "FROM Production.Product;", sourceConnection)
            Dim reader As SqlDataReader = commandSourceData.ExecuteReader

            ' Open the destination connection. In the real world you would
            ' not use SqlBulkCopy to move data from one table to the other
            ' in the same database. This is for demonstration purposes only.
            Using destinationConnection As SqlConnection = _
                New SqlConnection(connectionString)
                destinationConnection.Open()

                ' Set up the bulk copy object.
                ' The column positions in the source data reader
                ' match the column positions in the destination table,
                ' so there is no need to map columns.
                Using bulkCopy As SqlBulkCopy = _
                  New SqlBulkCopy(destinationConnection)
                    bulkCopy.DestinationTableName = _
                    "dbo.BulkCopyDemoMatchingColumns"

                    Try
                        ' Write from the source to the destination.
                        bulkCopy.WriteToServer(reader)

                    Catch ex As Exception
                        Console.WriteLine(ex.Message)

                    Finally
                        ' Close the SqlDataReader. The SqlBulkCopy
```

```
                        ' object is automatically closed at the end
                        ' of the Using block.
                        reader.Close()
                    End Try
                End Using

                ' Perform a final count on the destination table
                ' to see how many rows were added.
                Dim countEnd As Long = _
                    System.Convert.ToInt32(commandRowCount.ExecuteScalar())
                Console.WriteLine("Ending row count = {0}", countEnd)
                Console.WriteLine("{0} rows were added.", countEnd - countStart)

                Console.WriteLine("Press Enter to finish.")
                Console.ReadLine()
            End Using
        End Using
    End Sub

    Private Function GetConnectionString() As String
        ' To avoid storing the sourceConnection string in your code,
        ' you can retrieve it from a configuration file.
        Return "Data Source=(local);" & _
            "Integrated Security=true;" & _
            "Initial Catalog=AdventureWorks;"
    End Function
End Module
```

Performing a Bulk Copy Operation Using Transact-SQL and the Command Class

The following example illustrates how to use the ExecuteNonQuery method to execute the BULK INSERT statement.

> **NOTE**
>
> The file path for the data source is relative to the server. The server process must have access to that path in order for the bulk copy operation to succeed.

```
Using connection As SqlConnection = New SqlConnection(connectionString)
Dim queryString As String = _
    "BULK INSERT Northwind.dbo.[Order Details] FROM " & _
    "'f:\mydata\data.tbl' WITH (FORMATFILE='f:\mydata\data.fmt' )"
connection.Open()
SqlCommand command = New SqlCommand(queryString, connection);

command.ExecuteNonQuery()
End Using
```

```
using (SqlConnection connection = New SqlConnection(connectionString))
{
string queryString =  "BULK INSERT Northwind.dbo.[Order Details] " +
    "FROM 'f:\mydata\data.tbl' " +
    "WITH ( FORMATFILE='f:\mydata\data.fmt' )";
connection.Open();
SqlCommand command = new SqlCommand(queryString, connection);

command.ExecuteNonQuery();
}
```

See also

- Bulk Copy Operations in SQL Server
- ADO.NET Overview

Multiple Bulk Copy Operations

4/26/2022 • 6 minutes to read • Edit Online

You can perform multiple bulk copy operations using a single instance of a SqlBulkCopy class. If the operation parameters change between copies (for example, the name of the destination table), you must update them prior to any subsequent calls to any of the **WriteToServer** methods, as demonstrated in the following example. Unless explicitly changed, all property values remain the same as they were on the previous bulk copy operation for a given instance.

> **NOTE**
>
> Performing multiple bulk copy operations using the same instance of SqlBulkCopy is usually more efficient than using a separate instance for each operation.

If you perform several bulk copy operations using the same SqlBulkCopy object, there are no restrictions on whether source or target information is equal or different in each operation. However, you must ensure that column association information is properly set each time you write to the server.

> **IMPORTANT**
>
> This sample will not run unless you have created the work tables as described in Bulk Copy Example Setup. This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT … SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a connection to the AdventureWorks database.
        using (SqlConnection connection =
                   new SqlConnection(connectionString))
        {
            connection.Open();

            // Empty the destination tables.
            SqlCommand deleteHeader = new SqlCommand(
                "DELETE FROM dbo.BulkCopyDemoOrderHeader;",
                connection);
            deleteHeader.ExecuteNonQuery();
            SqlCommand deleteDetail = new SqlCommand(
                "DELETE FROM dbo.BulkCopyDemoOrderDetail;",
                connection);
            deleteDetail.ExecuteNonQuery();

            // Perform an initial count on the destination
            //  table with matching columns.
            SqlCommand countRowHeader = new SqlCommand(
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderHeader;",
                connection);
            long countStartHeader = System.Convert.ToInt32(
                countRowHeader.ExecuteScalar());
            Console.WriteLine(
                "Starting row count for Header table: {0}",
```

```
        "Starting row count for Header table = {0}",
        countStartHeader);

    // Perform an initial count on the destination
    // table with different column positions.
    SqlCommand countRowDetail = new SqlCommand(
        "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderDetail;",
        connection);
    long countStartDetail = System.Convert.ToInt32(
        countRowDetail.ExecuteScalar());
    Console.WriteLine(
        "Starting row count for Detail table = {0}",
        countStartDetail);

    // Get data from the source table as a SqlDataReader.
    // The Sales.SalesOrderHeader and Sales.SalesOrderDetail
    // tables are quite large and could easily cause a timeout
    // if all data from the tables is added to the destination.
    // To keep the example simple and quick, a parameter is
    // used to select only orders for a particular account
    // as the source for the bulk insert.
    SqlCommand headerData = new SqlCommand(
        "SELECT [SalesOrderID], [OrderDate], " +
        "[AccountNumber] FROM [Sales].[SalesOrderHeader] " +
        "WHERE [AccountNumber] = @accountNumber;",
        connection);
    SqlParameter parameterAccount = new SqlParameter();
    parameterAccount.ParameterName = "@accountNumber";
    parameterAccount.SqlDbType = SqlDbType.NVarChar;
    parameterAccount.Direction = ParameterDirection.Input;
    parameterAccount.Value = "10-4020-000034";
    headerData.Parameters.Add(parameterAccount);
    SqlDataReader readerHeader = headerData.ExecuteReader();

    // Get the Detail data in a separate connection.
    using (SqlConnection connection2 = new SqlConnection(connectionString))
    {
        connection2.Open();
        SqlCommand sourceDetailData = new SqlCommand(
            "SELECT [Sales].[SalesOrderDetail].[SalesOrderID], [SalesOrderDetailID], " +
            "[OrderQty], [ProductID], [UnitPrice] FROM [Sales].[SalesOrderDetail] " +
            "INNER JOIN [Sales].[SalesOrderHeader] ON [Sales].[SalesOrderDetail]." +
            "[SalesOrderID] = [Sales].[SalesOrderHeader].[SalesOrderID] " +
            "WHERE [AccountNumber] = @accountNumber;", connection2);

        SqlParameter accountDetail = new SqlParameter();
        accountDetail.ParameterName = "@accountNumber";
        accountDetail.SqlDbType = SqlDbType.NVarChar;
        accountDetail.Direction = ParameterDirection.Input;
        accountDetail.Value = "10-4020-000034";
        sourceDetailData.Parameters.Add(accountDetail);
        SqlDataReader readerDetail = sourceDetailData.ExecuteReader();

        // Create the SqlBulkCopy object.
        using (SqlBulkCopy bulkCopy =
                    new SqlBulkCopy(connectionString))
        {
            bulkCopy.DestinationTableName =
                "dbo.BulkCopyDemoOrderHeader";

            // Guarantee that columns are mapped correctly by
            // defining the column mappings for the order.
            bulkCopy.ColumnMappings.Add("SalesOrderID", "SalesOrderID");
            bulkCopy.ColumnMappings.Add("OrderDate", "OrderDate");
            bulkCopy.ColumnMappings.Add("AccountNumber", "AccountNumber");

            // Write readerHeader to the destination.
            try
            {
```

```
                    bulkCopy.WriteToServer(readerHeader);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
                finally
                {
                    readerHeader.Close();
                }

                // Set up the order details destination.
                bulkCopy.DestinationTableName ="dbo.BulkCopyDemoOrderDetail";

                // Clear the ColumnMappingCollection.
                bulkCopy.ColumnMappings.Clear();

                // Add order detail column mappings.
                bulkCopy.ColumnMappings.Add("SalesOrderID", "SalesOrderID");
                bulkCopy.ColumnMappings.Add("SalesOrderDetailID", "SalesOrderDetailID");
                bulkCopy.ColumnMappings.Add("OrderQty", "OrderQty");
                bulkCopy.ColumnMappings.Add("ProductID", "ProductID");
                bulkCopy.ColumnMappings.Add("UnitPrice", "UnitPrice");

                // Write readerDetail to the destination.
                try
                {
                    bulkCopy.WriteToServer(readerDetail);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
                finally
                {
                    readerDetail.Close();
                }
            }

            // Perform a final count on the destination
            // tables to see how many rows were added.
            long countEndHeader = System.Convert.ToInt32(
                countRowHeader.ExecuteScalar());
            Console.WriteLine("{0} rows were added to the Header table.",
                countEndHeader - countStartHeader);
            long countEndDetail = System.Convert.ToInt32(
                countRowDetail.ExecuteScalar());
            Console.WriteLine("{0} rows were added to the Detail table.",
                countEndDetail - countStartDetail);
            Console.WriteLine("Press Enter to finish.");
            Console.ReadLine();
        }
    }
}

private static string GetConnectionString()
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}
```

```
Imports System.Data.SqlClient

Module Module1
```

```
Module Module1
    Sub Main()
        Dim connectionString As String = GetConnectionString()

        ' Open a connection to the AdventureWorks database.
        Using connection As SqlConnection = New SqlConnection(connectionString)
            connection.Open()

            ' Empty the destination tables.
            Dim deleteHeader As New SqlCommand( _
              "DELETE FROM dbo.BulkCopyDemoOrderHeader;", connection)
            deleteHeader.ExecuteNonQuery()
            deleteHeader.Dispose()
            Dim deleteDetail As New SqlCommand( _
              "DELETE FROM dbo.BulkCopyDemoOrderDetail;", connection)
            deleteDetail.ExecuteNonQuery()

            ' Perform an initial count on the destination table
            ' with matching columns.
            Dim countRowHeader As New SqlCommand( _
               "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderHeader;", _
                connection)
            Dim countStartHeader As Long = System.Convert.ToInt32( _
             countRowHeader.ExecuteScalar())
            Console.WriteLine("Starting row count for Header table = {0}", _
             countStartHeader)

            ' Perform an initial count on the destination table
            ' with different column positions.
            Dim countRowDetail As New SqlCommand( _
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderDetail;", _
                connection)
            Dim countStartDetail As Long = System.Convert.ToInt32( _
                countRowDetail.ExecuteScalar())
            Console.WriteLine("Starting row count for Detail table = " & _
               countStartDetail)

            ' Get data from the source table as a SqlDataReader.
            ' The Sales.SalesOrderHeader and Sales.SalesOrderDetail
            ' tables are quite large and could easily cause a timeout
            ' if all data from the tables is added to the destination.
            ' To keep the example simple and quick, a parameter is
            ' used to select only orders for a particular account as
            ' the source for the bulk insert.
            Dim headerData As SqlCommand = New SqlCommand( _
             "SELECT [SalesOrderID], [OrderDate], " & _
             "[AccountNumber] FROM [Sales].[SalesOrderHeader] " & _
             "WHERE [AccountNumber] = @accountNumber;", _
             connection)

            Dim parameterAccount As SqlParameter = New SqlParameter()
            parameterAccount.ParameterName = "@accountNumber"
            parameterAccount.SqlDbType = SqlDbType.NVarChar
            parameterAccount.Direction = ParameterDirection.Input
            parameterAccount.Value = "10-4020-000034"
            headerData.Parameters.Add(parameterAccount)

            Dim readerHeader As SqlDataReader = _
             headerData.ExecuteReader()

            ' Get the Detail data in a separate connection.
            Using connection2 As SqlConnection = New SqlConnection(connectionString)
                connection2.Open()

                Dim sourceDetailData As SqlCommand = New SqlCommand( _
                 "SELECT [Sales].[SalesOrderDetail].[SalesOrderID], " & _
                 "[SalesOrderDetailID], [OrderQty], [ProductID], [UnitPrice] " & _
                 "FROM [Sales].[SalesOrderDetail] INNER JOIN " & _
                 "[Sales].[SalesOrderHeader] " & _
```

```
 "ON [Sales].[SalesOrderDetail].[SalesOrderID] = " & _
 "[Sales].[SalesOrderHeader].[SalesOrderID] " & _
 "WHERE [AccountNumber] = @accountNumber;", connection2)

Dim accountDetail As SqlParameter = New SqlParameter()
accountDetail.ParameterName = "@accountNumber"
accountDetail.SqlDbType = SqlDbType.NVarChar
accountDetail.Direction = ParameterDirection.Input
accountDetail.Value = "10-4020-000034"
sourceDetailData.Parameters.Add( _
 accountDetail)

Dim readerDetail As SqlDataReader = _
 sourceDetailData.ExecuteReader()

' Create the SqlBulkCopy object.
Using bulkCopy As SqlBulkCopy = _
  New SqlBulkCopy(connectionString)
    bulkCopy.DestinationTableName = "dbo.BulkCopyDemoOrderHeader"

    ' Guarantee that columns are mapped correctly by
    ' defining the column mappings for the order.
    bulkCopy.ColumnMappings.Add("SalesOrderID", "SalesOrderID")
    bulkCopy.ColumnMappings.Add("OrderDate", "OrderDate")
    bulkCopy.ColumnMappings.Add("AccountNumber", "AccountNumber")

    ' Write readerHeader to the destination.
    Try
        bulkCopy.WriteToServer(readerHeader)
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    Finally
        readerHeader.Close()
    End Try

    ' Set up the order details destination.
    bulkCopy.DestinationTableName = "dbo.BulkCopyDemoOrderDetail"

    ' Clear the ColumnMappingCollection.
    bulkCopy.ColumnMappings.Clear()

    ' Add order detail column mappings.
    bulkCopy.ColumnMappings.Add("SalesOrderID", "SalesOrderID")
    bulkCopy.ColumnMappings.Add("SalesOrderDetailID", "SalesOrderDetailID")
    bulkCopy.ColumnMappings.Add("OrderQty", "OrderQty")
    bulkCopy.ColumnMappings.Add("ProductID", "ProductID")
    bulkCopy.ColumnMappings.Add("UnitPrice", "UnitPrice")

    ' Write readerDetail to the destination.
    Try
        bulkCopy.WriteToServer(readerDetail)
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    Finally
        readerDetail.Close()
    End Try
End Using

' Perform a final count on the destination tables
' to see how many rows were added.
Dim countEndHeader As Long = System.Convert.ToInt32( _
  countRowHeader.ExecuteScalar())
Console.WriteLine("{0} rows were added to the Header table.", _
  countEndHeader - countStartHeader)
Dim countEndDetail As Long = System.Convert.ToInt32( _
   countRowDetail.ExecuteScalar())
Console.WriteLine("{0} rows were added to the Detail table.", _
    countEndDetail - countStartDetail)
```

```
                Console.WriteLine("Press Enter to finish.")
                Console.ReadLine()
            End Using
        End Using
    End Sub

    Private Function GetConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.
        Return "Data Source=(local);" & _
            "Integrated Security=true;" & _
            "Initial Catalog=AdventureWorks;"
    End Function
End Module
```

See also

- Bulk Copy Operations in SQL Server
- ADO.NET Overview

Transaction and Bulk Copy Operations

4/26/2022 • 17 minutes to read • Edit Online

Bulk copy operations can be performed as isolated operations or as part of a multiple step transaction. This latter option enables you to perform more than one bulk copy operation within the same transaction, as well as perform other database operations (such as inserts, updates, and deletes) while still being able to commit or roll back the entire transaction.

By default, a bulk copy operation is performed as an isolated operation. The bulk copy operation occurs in a non-transacted way, with no opportunity for rolling it back. If you need to roll back all or part of the bulk copy when an error occurs, you can use a SqlBulkCopy-managed transaction, perform the bulk copy operation within an existing transaction, or be enlisted in a **System.Transactions** Transaction.

Performing a Non-transacted Bulk Copy Operation

The following Console application shows what happens when a non-transacted bulk copy operation encounters an error partway through the operation.

In the example, the source table and destination table each include an `Identity` column named **ProductID**. The code first prepares the destination table by deleting all rows and then inserting a single row whose **ProductID** is known to exist in the source table. By default, a new value for the `Identity` column is generated in the destination table for each row added. In this example, an option is set when the connection is opened that forces the bulk load process to use the `Identity` values from the source table instead.

The bulk copy operation is executed with the BatchSize property set to 10. When the operation encounters the invalid row, an exception is thrown. In this first example, the bulk copy operation is non-transacted. All batches copied up to the point of the error are committed; the batch containing the duplicate key is rolled back, and the bulk copy operation is halted before processing any other batches.

> **NOTE**
>
> This sample will not run unless you have created the work tables as described in Bulk Copy Example Setup. This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT … SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
                   new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            //  Delete all from the destination table.
            SqlCommand commandDelete = new SqlCommand();
            commandDelete.Connection = sourceConnection;
            commandDelete.CommandText =
                "DELETE FROM dbo.BulkCopyDemoMatchingColumns";
            commandDelete.ExecuteNonQuery();
```

```
        //  Add a single row that will result in duplicate key
        //  when all rows from source are bulk copied.
        //  Note that this technique will only be successful in
        //  illustrating the point if a row with ProductID = 446
        //  exists in the AdventureWorks Production.Products table.
        //  If you have made changes to the data in this table, change
        //  the SQL statement in the code to add a ProductID that
        //  does exist in your version of the Production.Products
        //  table. Choose any ProductID in the middle of the table
        //  (not first or last row) to best illustrate the result.
        SqlCommand commandInsert = new SqlCommand();
        commandInsert.Connection = sourceConnection;
        commandInsert.CommandText =
            "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns ON;" +
            "INSERT INTO " + "dbo.BulkCopyDemoMatchingColumns " +
            "([ProductID], [Name] ,[ProductNumber]) " +
            "VALUES(446, 'Lock Nut 23','LN-3416');" +
            "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns OFF";
        commandInsert.ExecuteNonQuery();

        // Perform an initial count on the destination table.
        SqlCommand commandRowCount = new SqlCommand(
            "SELECT COUNT(*) FROM dbo.BulkCopyDemoMatchingColumns;",
            sourceConnection);
        long countStart = System.Convert.ToInt32(
            commandRowCount.ExecuteScalar());
        Console.WriteLine("Starting row count = {0}", countStart);

        //  Get data from the source table as a SqlDataReader.
        SqlCommand commandSourceData = new SqlCommand(
            "SELECT ProductID, Name, ProductNumber " +
            "FROM Production.Product;", sourceConnection);
        SqlDataReader reader = commandSourceData.ExecuteReader();

        // Set up the bulk copy object using the KeepIdentity option.
        using (SqlBulkCopy bulkCopy = new SqlBulkCopy(
                   connectionString, SqlBulkCopyOptions.KeepIdentity))
        {
            bulkCopy.BatchSize = 10;
            bulkCopy.DestinationTableName =
                "dbo.BulkCopyDemoMatchingColumns";

            // Write from the source to the destination.
            // This should fail with a duplicate key error
            // after some of the batches have been copied.
            try
            {
                bulkCopy.WriteToServer(reader);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                reader.Close();
            }
        }

        // Perform a final count on the destination
        // table to see how many rows were added.
        long countEnd = System.Convert.ToInt32(
            commandRowCount.ExecuteScalar());
        Console.WriteLine("Ending row count = {0}", countEnd);
        Console.WriteLine("{0} rows were added.", countEnd - countStart);
        Console.WriteLine("Press Enter to finish.");
        Console.ReadLine();
    }
```

```
    }

    private static string GetConnectionString()
        // To avoid storing the sourceConnection string in your code,
        // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}
```

```
Imports System.Data.SqlClient

Module Module1
    Sub Main()
        Dim connectionString As String = GetConnectionString()

        ' Open a sourceConnection to the AdventureWorks database.
        Using sourceConnection As SqlConnection = _
           New SqlConnection(connectionString)
            sourceConnection.Open()

            ' Delete all from the destination table.
            Dim commandDelete As New SqlCommand
            commandDelete.Connection = sourceConnection
            commandDelete.CommandText = _
               "DELETE FROM dbo.BulkCopyDemoMatchingColumns"
            commandDelete.ExecuteNonQuery()

            ' Add a single row that will result in duplicate key
            ' when all rows from source are bulk copied.
            ' Note that this technique will only be successful in
            ' illustrating the point if a row with ProductID = 446
            ' exists in the AdventureWorks Production.Products table.
            ' If you have made changes to the data in this table, change
            ' the SQL statement in the code to add a ProductID that
            ' does exist in your version of the Production.Products
            ' table. Choose any ProductID in the middle of the table
            ' (not first or last row) to best illustrate the result.
            Dim commandInsert As New SqlCommand
            commandInsert.Connection = sourceConnection
            commandInsert.CommandText = _
               "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns ON;" & _
               "INSERT INTO dbo.BulkCopyDemoMatchingColumns " & _
               "([ProductID], [Name] ,[ProductNumber]) " & _
               "VALUES(446, 'Lock Nut 23','LN-3416');" & _
               "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns OFF"
            commandInsert.ExecuteNonQuery()

            ' Perform an initial count on the destination table.
            Dim commandRowCount As New SqlCommand( _
               "SELECT COUNT(*) FROM dbo.BulkCopyDemoMatchingColumns;", _
                sourceConnection)
            Dim countStart As Long = _
               System.Convert.ToInt32(commandRowCount.ExecuteScalar())
            Console.WriteLine("Starting row count = {0}", countStart)

            ' Get data from the source table as a SqlDataReader.
            Dim commandSourceData As SqlCommand = New SqlCommand( _
               "SELECT ProductID, Name, ProductNumber " & _
               "FROM Production.Product;", sourceConnection)
            Dim reader As SqlDataReader = _
             commandSourceData.ExecuteReader()

            ' Set up the bulk copy object using the KeepIdentity option.
            Using bulkCopy As SqlBulkCopy = New SqlBulkCopy(connectionString, _
```

```
                SqlBulkCopyOptions.KeepIdentity)
                  bulkCopy.BatchSize = 10
                  bulkCopy.DestinationTableName = "dbo.BulkCopyDemoMatchingColumns"

                  ' Write from the source to the destination.
                  ' This should fail with a duplicate key error
                  ' after some of the batches have already been copied.
                  Try
                      bulkCopy.WriteToServer(reader)

                  Catch ex As Exception
                      Console.WriteLine(ex.Message)

                  Finally
                      reader.Close()
                  End Try
              End Using

              ' Perform a final count on the destination table
              ' to see how many rows were added.
              Dim countEnd As Long = _
                  System.Convert.ToInt32(commandRowCount.ExecuteScalar())
              Console.WriteLine("Ending row count = {0}", countEnd)
              Console.WriteLine("{0} rows were added.", countEnd - countStart)
              Console.WriteLine("Press Enter to finish.")
              Console.ReadLine()
          End Using
      End Sub

      Private Function GetConnectionString() As String
          ' To avoid storing the sourceConnection string in your code,
          ' you can retrieve it from a configuration file.
          Return "Data Source=(local);" & _
              "Integrated Security=true;" & _
              "Initial Catalog=AdventureWorks;"
      End Function
  End Module
```

Performing a Dedicated Bulk Copy Operation in a Transaction

By default, a bulk copy operation is its own transaction. When you want to perform a dedicated bulk copy operation, create a new instance of SqlBulkCopy with a connection string, or use an existing SqlConnection object without an active transaction. In each scenario, the bulk copy operation creates, and then commits or rolls back the transaction.

You can explicitly specify the UseInternalTransaction option in the SqlBulkCopy class constructor to explicitly cause a bulk copy operation to execute in its own transaction, causing each batch of the bulk copy operation to execute within a separate transaction.

> **NOTE**
> Since different batches are executed in different transactions, if an error occurs during the bulk copy operation, all the rows in the current batch will be rolled back, but rows from previous batches will remain in the database.

The following console application is similar to the previous example, with one exception: In this example, the bulk copy operation manages its own transactions. All batches copied up to the point of the error are committed; the batch containing the duplicate key is rolled back, and the bulk copy operation is halted before processing any other batches.

IMPORTANT

This sample will not run unless you have created the work tables as described in Bulk Copy Example Setup. This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT … SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
                   new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            //  Delete all from the destination table.
            SqlCommand commandDelete = new SqlCommand();
            commandDelete.Connection = sourceConnection;
            commandDelete.CommandText =
                "DELETE FROM dbo.BulkCopyDemoMatchingColumns";
            commandDelete.ExecuteNonQuery();

            //  Add a single row that will result in duplicate key
            //  when all rows from source are bulk copied.
            //  Note that this technique will only be successful in
            //  illustrating the point if a row with ProductID = 446
            //  exists in the AdventureWorks Production.Products table.
            //  If you have made changes to the data in this table, change
            //  the SQL statement in the code to add a ProductID that
            //  does exist in your version of the Production.Products
            //  table. Choose any ProductID in the middle of the table
            //  (not first or last row) to best illustrate the result.
            SqlCommand commandInsert = new SqlCommand();
            commandInsert.Connection = sourceConnection;
            commandInsert.CommandText =
                "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns ON;" +
                "INSERT INTO " + "dbo.BulkCopyDemoMatchingColumns " +
                "([ProductID], [Name] ,[ProductNumber]) " +
                "VALUES(446, 'Lock Nut 23','LN-3416');" +
                "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns OFF";
            commandInsert.ExecuteNonQuery();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoMatchingColumns;",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            //  Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader = commandSourceData.ExecuteReader();

            // Set up the bulk copy object.
            // Note that when specifying the UseInternalTransaction
            // option, you cannot also specify an external transaction.
            // Therefore, you must use the SqlBulkCopy construct that
            // requires a string for the connection, rather than an
```

```
            // existing SqlConnection object.
            using (SqlBulkCopy bulkCopy = new SqlBulkCopy(
                       connectionString, SqlBulkCopyOptions.KeepIdentity |
                       SqlBulkCopyOptions.UseInternalTransaction))
            {
                bulkCopy.BatchSize = 10;
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoMatchingColumns";

                // Write from the source to the destination.
                // This should fail with a duplicate key error
                // after some of the batches have been copied.
                try
                {
                    bulkCopy.WriteToServer(reader);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
                finally
                {
                    reader.Close();
                }
            }

            // Perform a final count on the destination
            // table to see how many rows were added.
            long countEnd = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Ending row count = {0}", countEnd);
            Console.WriteLine("{0} rows were added.", countEnd - countStart);
            Console.WriteLine("Press Enter to finish.");
            Console.ReadLine();
        }
    }

    private static string GetConnectionString()
        // To avoid storing the sourceConnection string in your code,
        // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}
```

```
Imports System.Data.SqlClient

Module Module1
    Sub Main()
        Dim connectionString As String = GetConnectionString()

        ' Open a sourceConnection to the AdventureWorks database.
        Using sourceConnection As SqlConnection = _
           New SqlConnection(connectionString)
            sourceConnection.Open()

            ' Delete all from the destination table.
            Dim commandDelete As New SqlCommand
            commandDelete.Connection = sourceConnection
            commandDelete.CommandText = _
               "DELETE FROM dbo.BulkCopyDemoMatchingColumns"
            commandDelete.ExecuteNonQuery()

            ' Add a single row that will result in duplicate key
            ' when all rows from source are bulk copied.
            ' Note that this technique will only be successful in
```

```
            ' Note that this technique will only be successful in
            ' illustrating the point if a row with ProductID = 446
            ' exists in the AdventureWorks Production.Products table.
            ' If you have made changes to the data in this table, change
            ' the SQL statement in the code to add a ProductID that
            ' does exist in your version of the Production.Products
            ' table. Choose any ProductID in the middle of the table
            ' (not first or last row) to best illustrate the result.
            Dim commandInsert As New SqlCommand
            commandInsert.Connection = sourceConnection
            commandInsert.CommandText = _
               "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns ON;" & _
               "INSERT INTO dbo.BulkCopyDemoMatchingColumns " & _
               "([ProductID], [Name] ,[ProductNumber]) " & _
               "VALUES(446, 'Lock Nut 23','LN-3416');" & _
               "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns OFF"
            commandInsert.ExecuteNonQuery()

            ' Perform an initial count on the destination table.
            Dim commandRowCount As New SqlCommand( _
               "SELECT COUNT(*) FROM dbo.BulkCopyDemoMatchingColumns;", _
                sourceConnection)
            Dim countStart As Long = _
               System.Convert.ToInt32(commandRowCount.ExecuteScalar())
            Console.WriteLine("Starting row count = {0}", countStart)

            ' Get data from the source table as a SqlDataReader.
            Dim commandSourceData As SqlCommand = New SqlCommand( _
               "SELECT ProductID, Name, ProductNumber " & _
               "FROM Production.Product;", sourceConnection)
            Dim reader As SqlDataReader = _
             commandSourceData.ExecuteReader()

            ' Set up the bulk copy object.
            ' Note that when specifying the UseInternalTransaction option,
            ' you cannot also specify an external transaction. Therefore,
            ' you must use the SqlBulkCopy construct that requires a string
            ' for the connection, rather than an existing SqlConnection object.
            Using bulkCopy As SqlBulkCopy = New SqlBulkCopy(connectionString, _
             SqlBulkCopyOptions.UseInternalTransaction Or _
             SqlBulkCopyOptions.KeepIdentity)
                bulkCopy.BatchSize = 10
                bulkCopy.DestinationTableName = "dbo.BulkCopyDemoMatchingColumns"

                ' Write from the source to the destination.
                ' This should fail with a duplicate key error
                ' after some of the batches have already been copied.
                Try
                    bulkCopy.WriteToServer(reader)

                Catch ex As Exception
                    Console.WriteLine(ex.Message)

                Finally
                    reader.Close()
                End Try
            End Using

            ' Perform a final count on the destination table
            ' to see how many rows were added.
            Dim countEnd As Long = _
                System.Convert.ToInt32(commandRowCount.ExecuteScalar())
            Console.WriteLine("Ending row count = {0}", countEnd)
            Console.WriteLine("{0} rows were added.", countEnd - countStart)
            Console.WriteLine("Press Enter to finish.")
            Console.ReadLine()
        End Using
    End Sub
```

```
    Private Function GetConnectionString() As String
        ' To avoid storing the sourceConnection string in your code,
        ' you can retrieve it from a configuration file.
        Return "Data Source=(local);" & _
            "Integrated Security=true;" & _
            "Initial Catalog=AdventureWorks;"
    End Function
End Module
```

Using Existing Transactions

You can specify an existing SqlTransaction object as a parameter in a SqlBulkCopy constructor. In this situation, the bulk copy operation is performed in an existing transaction, and no change is made to the transaction state (that is, it is neither committed nor aborted). This allows an application to include the bulk copy operation in a transaction with other database operations. However, if you do not specify a SqlTransaction object and pass a null reference, and the connection has an active transaction, an exception is thrown.

If you need to roll back the entire bulk copy operation because an error occurs, or if the bulk copy should execute as part of a larger process that can be rolled back, you can provide a SqlTransaction object to the SqlBulkCopy constructor.

The following console application is similar to the first (non-transacted) example, with one exception: in this example, the bulk copy operation is included in a larger, external transaction. When the primary key violation error occurs, the entire transaction is rolled back and no rows are added to the destination table.

> **IMPORTANT**
>
> This sample will not run unless you have created the work tables as described in Bulk Copy Example Setup. This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT … SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
                   new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            //  Delete all from the destination table.
            SqlCommand commandDelete = new SqlCommand();
            commandDelete.Connection = sourceConnection;
            commandDelete.CommandText =
                "DELETE FROM dbo.BulkCopyDemoMatchingColumns";
            commandDelete.ExecuteNonQuery();

            //  Add a single row that will result in duplicate key
            //  when all rows from source are bulk copied.
            //  Note that this technique will only be successful in
            //  illustrating the point if a row with ProductID = 446
            //  exists in the AdventureWorks Production.Products table.
            //  If you have made changes to the data in this table, change
            //  the SQL statement in the code to add a ProductID that
            //  does exist in your version of the Production.Products
            //  table. Choose any ProductID in the middle of the table
            //  (not first or last row) to best illustrate the result.
```

```
            SqlCommand commandInsert = new SqlCommand();
            commandInsert.Connection = sourceConnection;
            commandInsert.CommandText =
                "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns ON;" +
                "INSERT INTO " + "dbo.BulkCopyDemoMatchingColumns " +
                "([ProductID], [Name] ,[ProductNumber]) " +
                "VALUES(446, 'Lock Nut 23','LN-3416');" +
                "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns OFF";
            commandInsert.ExecuteNonQuery();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoMatchingColumns;",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            //  Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader = commandSourceData.ExecuteReader();

            //Set up the bulk copy object inside the transaction.
            using (SqlConnection destinationConnection =
                       new SqlConnection(connectionString))
            {
                destinationConnection.Open();

                using (SqlTransaction transaction =
                           destinationConnection.BeginTransaction())
                {
                    using (SqlBulkCopy bulkCopy = new SqlBulkCopy(
                               destinationConnection, SqlBulkCopyOptions.KeepIdentity,
                               transaction))
                    {
                        bulkCopy.BatchSize = 10;
                        bulkCopy.DestinationTableName =
                            "dbo.BulkCopyDemoMatchingColumns";

                        // Write from the source to the destination.
                        // This should fail with a duplicate key error.
                        try
                        {
                            bulkCopy.WriteToServer(reader);
                            transaction.Commit();
                        }
                        catch (Exception ex)
                        {
                            Console.WriteLine(ex.Message);
                            transaction.Rollback();
                        }
                        finally
                        {
                            reader.Close();
                        }
                    }
                }
            }

            // Perform a final count on the destination
            // table to see how many rows were added.
            long countEnd = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Ending row count = {0}", countEnd);
            Console.WriteLine("{0} rows were added.", countEnd - countStart);
            Console.WriteLine("Press Enter to finish.");
            Console.ReadLine();
```

```
        }
    }

    private static string GetConnectionString()
        // To avoid storing the sourceConnection string in your code,
        // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}
```

```
Imports System.Data.SqlClient

Module Module1
    Sub Main()
        Dim connectionString As String = GetConnectionString()

        ' Open a sourceConnection to the AdventureWorks database.
        Using sourceConnection As SqlConnection = _
           New SqlConnection(connectionString)
            sourceConnection.Open()

            ' Delete all from the destination table.
            Dim commandDelete As New SqlCommand
            commandDelete.Connection = sourceConnection
            commandDelete.CommandText = _
               "DELETE FROM dbo.BulkCopyDemoMatchingColumns"
            commandDelete.ExecuteNonQuery()

            ' Add a single row that will result in duplicate key
            ' when all rows from source are bulk copied.
            ' Note that this technique will only be successful in
            ' illustrating the point if a row with ProductID = 446
            ' exists in the AdventureWorks Production.Products table.
            ' If you have made changes to the data in this table, change
            ' the SQL statement in the code to add a ProductID that
            ' does exist in your version of the Production.Products
            ' table. Choose any ProductID in the middle of the table
            ' (not first or last row) to best illustrate the result.
            Dim commandInsert As New SqlCommand
            commandInsert.Connection = sourceConnection
            commandInsert.CommandText = _
               "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns ON;" & _
               "INSERT INTO dbo.BulkCopyDemoMatchingColumns " & _
               "([ProductID], [Name] ,[ProductNumber]) " & _
               "VALUES(446, 'Lock Nut 23','LN-3416');" & _
               "SET IDENTITY_INSERT dbo.BulkCopyDemoMatchingColumns OFF"
            commandInsert.ExecuteNonQuery()

            ' Perform an initial count on the destination table.
            Dim commandRowCount As New SqlCommand( _
               "SELECT COUNT(*) FROM dbo.BulkCopyDemoMatchingColumns;", _
                sourceConnection)
            Dim countStart As Long = _
               System.Convert.ToInt32(commandRowCount.ExecuteScalar())
            Console.WriteLine("Starting row count = {0}", countStart)

            ' Get data from the source table as a SqlDataReader.
            Dim commandSourceData As SqlCommand = New SqlCommand( _
               "SELECT ProductID, Name, ProductNumber " & _
               "FROM Production.Product;", sourceConnection)
            Dim reader As SqlDataReader = _
             commandSourceData.ExecuteReader()

            ' Set up the bulk copy object inside the transaction.
```

```
            Using destinationConnection As SqlConnection = _
               New SqlConnection(connectionString)
                destinationConnection.Open()

                Using transaction As SqlTransaction = _
                  destinationConnection.BeginTransaction()

                    Using bulkCopy As SqlBulkCopy = New _
                      SqlBulkCopy(destinationConnection, _
                         SqlBulkCopyOptions.KeepIdentity, transaction)
                        bulkCopy.BatchSize = 10
                        bulkCopy.DestinationTableName = _
                         "dbo.BulkCopyDemoMatchingColumns"

                        ' Write from the source to the destination.
                        ' This should fail with a duplicate key error.
                        Try
                            bulkCopy.WriteToServer(reader)
                            transaction.Commit()

                        Catch ex As Exception
                            Console.WriteLine(ex.Message)
                            transaction.Rollback()

                        Finally
                            reader.Close()
                        End Try
                    End Using
                End Using
            End Using

            ' Perform a final count on the destination table
            ' to see how many rows were added.
            Dim countEnd As Long = _
                System.Convert.ToInt32(commandRowCount.ExecuteScalar())
            Console.WriteLine("Ending row count = {0}", countEnd)
            Console.WriteLine("{0} rows were added.", countEnd - countStart)
            Console.WriteLine("Press Enter to finish.")
            Console.ReadLine()
        End Using
    End Sub

    Private Function GetConnectionString() As String
        ' To avoid storing the sourceConnection string in your code,
        ' you can retrieve it from a configuration file.
        Return "Data Source=(local);" & _
            "Integrated Security=true;" & _
            "Initial Catalog=AdventureWorks;"
    End Function
End Module
```

See also

- Bulk Copy Operations in SQL Server
- ADO.NET Overview

Multiple Active Result Sets (MARS)

4/26/2022 • 2 minutes to read • Edit Online

Multiple Active Result Sets (MARS) is a feature that allows the execution of multiple batches on a single connection. In previous versions, only one batch could be executed at a time against a single connection. Executing multiple batches with MARS does not imply simultaneous execution of operations.

In This Section

Enabling Multiple Active Result Sets
Discusses how to use MARS with SQL Server.

Manipulating Data
Provides examples of coding MARS applications.

Related Sections

Asynchronous Operations
Provides details on using the new asynchronous features in ADO.NET.

See also

- SQL Server and ADO.NET
- ADO.NET Overview

Enabling Multiple Active Result Sets

4/26/2022 • 5 minutes to read • Edit Online

Multiple Active Result Sets (MARS) is a feature that works with SQL Server to allow the execution of multiple batches on a single connection. When MARS is enabled for use with SQL Server, each command object used adds a session to the connection.

> **NOTE**
>
> A single MARS session opens one logical connection for MARS to use and then one logical connection for each active command.

Enabling and Disabling MARS in the Connection String

> **NOTE**
>
> The following connection strings use the sample **AdventureWorks** database included with SQL Server. The connection strings provided assume that the database is installed on a server named MSSQL1. Modify the connection string as necessary for your environment.

The MARS feature is disabled by default. It can be enabled by adding the "MultipleActiveResultSets=True" keyword pair to your connection string. "True" is the only valid value for enabling MARS. The following example demonstrates how to connect to an instance of SQL Server and how to specify that MARS should be enabled.

```
Dim connectionString As String = "Data Source=MSSQL1;" & _
    "Initial Catalog=AdventureWorks;Integrated Security=SSPI;" & _
    "MultipleActiveResultSets=True"
```

```
string connectionString = "Data Source=MSSQL1;" +
    "Initial Catalog=AdventureWorks;Integrated Security=SSPI;" +
    "MultipleActiveResultSets=True";
```

You can disable MARS by adding the "MultipleActiveResultSets=False" keyword pair to your connection string. "False" is the only valid value for disabling MARS. The following connection string demonstrates how to disable MARS.

```
Dim connectionString As String = "Data Source=MSSQL1;" & _
    "Initial Catalog=AdventureWorks;Integrated Security=SSPI;" & _
    "MultipleActiveResultSets=False"
```

```
string connectionString = "Data Source=MSSQL1;" +
    "Initial Catalog=AdventureWorks;Integrated Security=SSPI;" +
    "MultipleActiveResultSets=False";
```

Special Considerations When Using MARS

In general, existing applications should not need modification to use a MARS-enabled connection. However, if

you wish to use MARS features in your applications, you should understand the following special considerations.

Statement Interleaving

MARS operations execute synchronously on the server. Statement interleaving of SELECT and BULK INSERT statements is allowed. However, data manipulation language (DML) and data definition language (DDL) statements execute atomically. Any statements attempting to execute while an atomic batch is executing are blocked. Parallel execution at the server is not a MARS feature.

If two batches are submitted under a MARS connection, one of them containing a SELECT statement, the other containing a DML statement, the DML can begin execution within execution of the SELECT statement. However, the DML statement must run to completion before the SELECT statement can make progress. If both statements are running under the same transaction, any changes made by a DML statement after the SELECT statement has started execution are not visible to the read operation.

A WAITFOR statement inside a SELECT statement does not yield the transaction while it is waiting, that is, until the first row is produced. This implies that no other batches can execute within the same connection while a WAITFOR statement is waiting.

MARS Session Cache

When a connection is opened with MARS enabled, a logical session is created, which adds additional overhead. To minimize overhead and enhance performance, **SqlClient** caches the MARS session within a connection. The cache contains at most 10 MARS sessions. This value is not user adjustable. If the session limit is reached, a new session is created—an error is not generated. The cache and sessions contained in it are per-connection; they are not shared across connections. When a session is released, it is returned to the pool unless the pool's upper limit has been reached. If the cache pool is full, the session is closed. MARS sessions do not expire. They are only cleaned up when the connection object is disposed. The MARS session cache is not preloaded. It is loaded as the application requires more sessions.

Thread Safety

MARS operations are not thread-safe.

Connection Pooling

MARS-enabled connections are pooled like any other connection. If an application opens two connections, one with MARS enabled and one with MARS disabled, the two connections are in separate pools. For more information, see SQL Server Connection Pooling (ADO.NET).

SQL Server Batch Execution Environment

When a connection is opened, a default environment is defined. This environment is then copied into a logical MARS session.

The batch execution environment includes the following components:

- Set options (for example, ANSI_NULLS, DATE_FORMAT, LANGUAGE, TEXTSIZE)
- Security context (user/application role)
- Database context (current database)
- Execution state variables (for example, @@ERROR, @@ROWCOUNT, @@FETCH_STATUS @@IDENTITY)
- Top-level temporary tables

With MARS, a default execution environment is associated to a connection. Every new batch that starts executing under a given connection receives a copy of the default environment. Whenever code is executed under a given batch, all changes made to the environment are scoped to the specific batch. Once execution finishes, the execution settings are copied into the default environment. In the case of a single batch issuing

several commands to be executed sequentially under the same transaction, semantics are the same as those exposed by connections involving earlier clients or servers.

Parallel Execution

MARS is not designed to remove all requirements for multiple connections in an application. If an application needs true parallel execution of commands against a server, multiple connections should be used.

For example, consider the following scenario. Two command objects are created, one for processing a result set and another for updating data; they share a common connection via MARS. In this scenario, the `Transaction`. `Commit` fails on the update until all the results have been read on the first command object, yielding the following exception:

Message: Transaction context in use by another session.

Source: .NET SqlClient Data Provider

Expected: (null)

Received: System.Data.SqlClient.SqlException

There are three options for handling this scenario:

1. Start the transaction after the reader is created, so that it is not part of the transaction. Every update then becomes its own transaction.
2. Commit all work after the reader is closed. This has the potential for a substantial batch of updates.
3. Don't use MARS; instead use a separate connection for each command object as you would have before MARS.

Detecting MARS Support

An application can check for MARS support by reading the `SqlConnection.ServerVersion` value. The major number should be 9 for SQL Server 2005 and 10 for SQL Server 2008.

See also

- Multiple Active Result Sets (MARS)
- ADO.NET Overview

Manipulating Data

4/26/2022 • 7 minutes to read • Edit Online

Before the introduction of Multiple Active Result Sets (MARS), developers had to use either multiple connections or server-side cursors to solve certain scenarios. In addition, when multiple connections were used in a transactional situation, bound connections (with **sp_getbindtoken** and **sp_bindsession**) were required. The following scenarios show how to use a MARS-enabled connection instead of multiple connections.

Using Multiple Commands with MARS

The following Console application demonstrates how to use two SqlDataReader objects with two SqlCommand objects and a single SqlConnection object with MARS enabled.

Example

The example opens a single connection to the **AdventureWorks** database. Using a SqlCommand object, a SqlDataReader is created. As the reader is used, a second SqlDataReader is opened, using data from the first SqlDataReader as input to the WHERE clause for the second reader.

> **NOTE**
>
> The following example uses the sample **AdventureWorks** database included with SQL Server. The connection string provided in the sample code assumes that the database is installed and available on the local computer. Modify the connection string as necessary for your environment.

```
Option Strict On
Option Explicit On

Imports System
Imports System.Data
Imports System.Data.SqlClient
Module Module1
  Sub Main()
    ' By default, MARS is disabled when connecting
    ' to a MARS-enabled host.
    ' It must be enabled in the connection string.
    Dim connectionString As String = GetConnectionString()

    Dim vendorID As Integer

    Dim vendorCmd As SqlCommand
    Dim productCmd As SqlCommand
    Dim productReader As SqlDataReader

    Dim vendorSQL As String = & _
      "SELECT VendorId, Name FROM Purchasing.Vendor"
    Dim productSQL As String = _
        "SELECT Production.Product.Name FROM Production.Product " & _
        "INNER JOIN Purchasing.ProductVendor " & _
        "ON Production.Product.ProductID = " & _
        "Purchasing.ProductVendor.ProductID " & _
        "WHERE Purchasing.ProductVendor.VendorID = @VendorId"

    Using awConnection As New SqlConnection(connectionString)
      vendorCmd = New SqlCommand(vendorSQL, awConnection)
      productCmd = New SqlCommand(productSQL, awConnection)
      productCmd.Parameters.Add("@VendorId", SqlDbType.Int)

      awConnection.Open()
      Using vendorReader As SqlDataReader = vendorCmd.ExecuteReader()
        While vendorReader.Read()
          Console.WriteLine(vendorReader("Name"))

          vendorID = CInt(vendorReader("VendorId"))

          productCmd.Parameters("@VendorId").Value = vendorID

          ' The following line of code requires
          ' a MARS-enabled connection.
          productReader = productCmd.ExecuteReader()
          Using productReader
            While productReader.Read()
              Console.WriteLine("  " & CStr(productReader("Name")))
            End While
          End Using
        End While
      End Using
    End Using

    Console.WriteLine("Press any key to continue")
    Console.ReadLine()
  End Sub

  Function GetConnectionString() As String
    ' To avoid storing the connection string in your code,
    ' you can retrieve it from a configuration file.
    Return "Data Source=(local);Integrated Security=SSPI;" & _
      "Initial Catalog=AdventureWorks; MultipleActiveResultSets=True"
  End Function
End Module
```

```
using System;
using System.Data;
using System.Data.SqlClient;

class Class1
{
static void Main()
{
  // By default, MARS is disabled when connecting
  // to a MARS-enabled host.
  // It must be enabled in the connection string.
  string connectionString = GetConnectionString();

  int vendorID;
  SqlDataReader productReader = null;
  string vendorSQL =
    "SELECT VendorId, Name FROM Purchasing.Vendor";
  string productSQL =
    "SELECT Production.Product.Name FROM Production.Product " +
    "INNER JOIN Purchasing.ProductVendor " +
    "ON Production.Product.ProductID = " +
    "Purchasing.ProductVendor.ProductID " +
    "WHERE Purchasing.ProductVendor.VendorID = @VendorId";

  using (SqlConnection awConnection =
    new SqlConnection(connectionString))
  {
    SqlCommand vendorCmd = new SqlCommand(vendorSQL, awConnection);
    SqlCommand productCmd =
      new SqlCommand(productSQL, awConnection);

    productCmd.Parameters.Add("@VendorId", SqlDbType.Int);

    awConnection.Open();
    using (SqlDataReader vendorReader = vendorCmd.ExecuteReader())
    {
      while (vendorReader.Read())
      {
        Console.WriteLine(vendorReader["Name"]);

        vendorID = (int)vendorReader["VendorId"];

        productCmd.Parameters["@VendorId"].Value = vendorID;
        // The following line of code requires
        // a MARS-enabled connection.
        productReader = productCmd.ExecuteReader();
        using (productReader)
        {
          while (productReader.Read())
          {
            Console.WriteLine("  " +
              productReader["Name"].ToString());
          }
        }
      }
  }
      Console.WriteLine("Press any key to continue");
      Console.ReadLine();
    }
  }
  private static string GetConnectionString()
  {
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
    return "Data Source=(local);Integrated Security=SSPI;" +
      "Initial Catalog=AdventureWorks;MultipleActiveResultSets=True";
  }
}
```

Reading and Updating Data with MARS

MARS allows a connection to be used for both read operations and data manipulation language (DML) operations with more than one pending operation. This feature eliminates the need for an application to deal with connection-busy errors. In addition, MARS can replace the use of server-side cursors, which generally consume more resources. Finally, because multiple operations can operate on a single connection, they can share the same transaction context, eliminating the need to use **sp_getbindtoken** and **sp_bindsession** system stored procedures.

Example

The following Console application demonstrates how to use two SqlDataReader objects with three SqlCommand objects and a single SqlConnection object with MARS enabled. The first command object retrieves a list of vendors whose credit rating is 5. The second command object uses the vendor ID provided from a SqlDataReader to load the second SqlDataReader with all of the products for the particular vendor. Each product record is visited by the second SqlDataReader. A calculation is performed to determine what the new **OnOrderQty** should be. The third command object is then used to update the **ProductVendor** table with the new value. This entire process takes place within a single transaction, which is rolled back at the end.

> **NOTE**
>
> The following example uses the sample **AdventureWorks** database included with SQL Server. The connection string provided in the sample code assumes that the database is installed and available on the local computer. Modify the connection string as necessary for your environment.

```
Option Strict On
Option Explicit On

Imports System
Imports System.Data
Imports System.Data.SqlClient

Module Module1

  Sub Main()
    ' By default, MARS is disabled when connecting
    ' to a MARS-enabled host.
    ' It must be enabled in the connection string.
    Dim connectionString As String = GetConnectionString()

    Dim updateTx As SqlTransaction
    Dim vendorCmd As SqlCommand
    Dim prodVendCmd As SqlCommand
    Dim updateCmd As SqlCommand

    Dim prodVendReader As SqlDataReader

    Dim vendorID As Integer
    Dim productID As Integer
    Dim minOrderQty As Integer
    Dim maxOrderQty As Integer
    Dim onOrderQty As Integer
    Dim recordsUpdated As Integer
    Dim totalRecordsUpdated As Integer

    Dim vendorSQL As String = _
        "SELECT VendorID, Name FROM Purchasing.Vendor " & _
        "WHERE CreditRating = 5"
    Dim prodVendSQL As String = _
        "SELECT ProductID, MaxOrderQty, MinOrderQty, OnOrderQty " & _
```

```
        "FROM Purchasing.ProductVendor " & _
        "WHERE VendorID = @VendorID"
    Dim updateSQL As String = _
        "UPDATE Purchasing.ProductVendor " & _
        "SET OnOrderQty = @OrderQty " & _
        "WHERE ProductID = @ProductID AND VendorID = @VendorID"

    Using awConnection As New SqlConnection(connectionString)
      awConnection.Open()
      updateTx = awConnection.BeginTransaction()

      vendorCmd = New SqlCommand(vendorSQL, awConnection)
      vendorCmd.Transaction = updateTx

      prodVendCmd = New SqlCommand(prodVendSQL, awConnection)
      prodVendCmd.Transaction = updateTx
      prodVendCmd.Parameters.Add("@VendorId", SqlDbType.Int)

      updateCmd = New SqlCommand(updateSQL, awConnection)
      updateCmd.Transaction = updateTx
      updateCmd.Parameters.Add("@OrderQty", SqlDbType.Int)
      updateCmd.Parameters.Add("@ProductID", SqlDbType.Int)
      updateCmd.Parameters.Add("@VendorID", SqlDbType.Int)

      Using vendorReader As SqlDataReader = vendorCmd.ExecuteReader()
        While vendorReader.Read()
          Console.WriteLine(vendorReader("Name"))

          vendorID = CInt(vendorReader("VendorID"))
          prodVendCmd.Parameters("@VendorID").Value = vendorID
          prodVendReader = prodVendCmd.ExecuteReader()

          Using prodVendReader
            While (prodVendReader.Read)
              productID = CInt(prodVendReader("ProductID"))

              If IsDBNull(prodVendReader("OnOrderQty")) Then
                minOrderQty = CInt(prodVendReader("MinOrderQty"))
                onOrderQty = minOrderQty
              Else
                maxOrderQty = CInt(prodVendReader("MaxOrderQty"))
                onOrderQty = CInt(maxOrderQty / 2)
              End If

              updateCmd.Parameters("@OrderQty").Value = onOrderQty
              updateCmd.Parameters("@ProductID").Value = productID
              updateCmd.Parameters("@VendorID").Value = vendorID

              recordsUpdated = updateCmd.ExecuteNonQuery()
              totalRecordsUpdated += recordsUpdated
            End While
          End Using
        End While
      End Using

      Console.WriteLine("Total Records Updated: " & _
        CStr(totalRecordsUpdated))
      updateTx.Rollback()
      Console.WriteLine("Transaction Rolled Back")
    End Using

    Console.WriteLine("Press any key to continue")
    Console.ReadLine()

  End Sub

  Function GetConnectionString() As String
    ' To avoid storing the connection string in your code,
    ' you can retrieve it from a configuration file.
```

```
    Return "Data Source=(local);Integrated Security=SSPI;" & _
      "Initial Catalog=AdventureWorks;MultipleActiveResultSets=True"
  End Function
End Module
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

class Program
{
static void Main()
{
  // By default, MARS is disabled when connecting
  // to a MARS-enabled host.
  // It must be enabled in the connection string.
  string connectionString = GetConnectionString();

  SqlTransaction updateTx = null;
  SqlCommand vendorCmd = null;
  SqlCommand prodVendCmd = null;
  SqlCommand updateCmd = null;

  SqlDataReader prodVendReader = null;

  int vendorID = 0;
  int productID = 0;
  int minOrderQty = 0;
  int maxOrderQty = 0;
  int onOrderQty = 0;
  int recordsUpdated = 0;
  int totalRecordsUpdated = 0;

  string vendorSQL =
      "SELECT VendorID, Name FROM Purchasing.Vendor " +
      "WHERE CreditRating = 5";
  string prodVendSQL =
      "SELECT ProductID, MaxOrderQty, MinOrderQty, OnOrderQty " +
      "FROM Purchasing.ProductVendor " +
      "WHERE VendorID = @VendorID";
  string updateSQL =
      "UPDATE Purchasing.ProductVendor " +
      "SET OnOrderQty = @OrderQty " +
      "WHERE ProductID = @ProductID AND VendorID = @VendorID";

  using (SqlConnection awConnection =
    new SqlConnection(connectionString))
  {
    awConnection.Open();
    updateTx = awConnection.BeginTransaction();

    vendorCmd = new SqlCommand(vendorSQL, awConnection);
    vendorCmd.Transaction = updateTx;

    prodVendCmd = new SqlCommand(prodVendSQL, awConnection);
    prodVendCmd.Transaction = updateTx;
    prodVendCmd.Parameters.Add("@VendorId", SqlDbType.Int);

    updateCmd = new SqlCommand(updateSQL, awConnection);
    updateCmd.Transaction = updateTx;
    updateCmd.Parameters.Add("@OrderQty", SqlDbType.Int);
    updateCmd.Parameters.Add("@ProductID", SqlDbType.Int);
    updateCmd.Parameters.Add("@VendorID", SqlDbType.Int);

    using (SqlDataReader vendorReader = vendorCmd.ExecuteReader())
```

```
    {
      while (vendorReader.Read())
      {
        Console.WriteLine(vendorReader["Name"]);

        vendorID = (int) vendorReader["VendorID"];
        prodVendCmd.Parameters["@VendorID"].Value = vendorID;
        prodVendReader = prodVendCmd.ExecuteReader();

        using (prodVendReader)
        {
          while (prodVendReader.Read())
          {
            productID = (int) prodVendReader["ProductID"];

            if (prodVendReader["OnOrderQty"] == DBNull.Value)
            {
              minOrderQty = (int) prodVendReader["MinOrderQty"];
              onOrderQty = minOrderQty;
            }
            else
            {
              maxOrderQty = (int) prodVendReader["MaxOrderQty"];
              onOrderQty = (int)(maxOrderQty / 2);
            }

            updateCmd.Parameters["@OrderQty"].Value = onOrderQty;
            updateCmd.Parameters["@ProductID"].Value = productID;
            updateCmd.Parameters["@VendorID"].Value = vendorID;

            recordsUpdated = updateCmd.ExecuteNonQuery();
            totalRecordsUpdated += recordsUpdated;
          }
        }
      }
    }
    Console.WriteLine("Total Records Updated: " +
      totalRecordsUpdated.ToString());
    updateTx.Rollback();
    Console.WriteLine("Transaction Rolled Back");
  }

  Console.WriteLine("Press any key to continue");
  Console.ReadLine();
}
private static string GetConnectionString()
{
  // To avoid storing the connection string in your code,
  // you can retrieve it from a configuration file.
  return "Data Source=(local);Integrated Security=SSPI;" +
    "Initial Catalog=AdventureWorks;" +
    "MultipleActiveResultSets=True";
  }
}
```

See also

- Multiple Active Result Sets (MARS)
- ADO.NET Overview

Asynchronous Operations

4/26/2022 • 2 minutes to read • Edit Online

Some database operations, such as command executions, can take significant time to complete. In such a case, single-threaded applications must block other operations and wait for the command to finish before they can continue their own operations. In contrast, being able to assign the long-running operation to a background thread allows the foreground thread to remain active throughout the operation. In a Windows application, for example, delegating the long-running operation to a background thread allows the user interface thread to remain responsive while the operation is executing.

The .NET Framework provides several standard asynchronous design patterns that developers can use to take advantage of background threads and free the user interface or high-priority threads to complete other operations. ADO.NET supports these same design patterns in its SqlCommand class. Specifically, the BeginExecuteNonQuery, BeginExecuteReader, and BeginExecuteXmlReader methods, paired with the EndExecuteNonQuery, EndExecuteReader, and EndExecuteXmlReader methods, provide the asynchronous support.

> **NOTE**
> Asynchronous programming is a core feature of the .NET Framework, and ADO.NET takes full advantage of the standard design patterns. For more information about the different asynchronous techniques available to developers, see Calling Synchronous Methods Asynchronously.

Although using asynchronous techniques with ADO.NET features does not add any special considerations, it is likely that more developers will use asynchronous features in ADO.NET than in other areas of the .NET Framework. It is important to be aware of the benefits and pitfalls of creating multithreaded applications. The examples that follow in this section point out several important issues that developers will need to take into account when building applications that incorporate multithreaded functionality.

In This Section

Windows Applications Using Callbacks
Provides an example demonstrating how to execute an asynchronous command safely, correctly handling interaction with a form and its contents from a separate thread.

ASP.NET Applications Using Wait Handles
Provides an example demonstrating how to execute multiple concurrent commands from an ASP.NET page, using Wait handles to manage the operation at completion of all the commands.

Polling in Console Applications
Provides an example demonstrating the use of polling to wait for the completion of an asynchronous command execution from a console application. This technique is also valid in a class library or other application without a user interface.

See also

- SQL Server and ADO.NET
- Calling Synchronous Methods Asynchronously
- ADO.NET Overview

Windows Applications Using Callbacks

4/26/2022 • 9 minutes to read • Edit Online

In most asynchronous processing scenarios, you want to start a database operation and continue running other processes without waiting for the database operation to complete. However, many scenarios require doing something once the database operation has ended. In a Windows application, for example, you may want to delegate the long-running operation to a background thread while allowing the user interface thread to remain responsive. However, when the database operation is complete, you want to use the results to populate the form. This type of scenario is best implemented with a callback.

You define a callback by specifying an AsyncCallback delegate in the BeginExecuteNonQuery, BeginExecuteReader, or BeginExecuteXmlReader method. The delegate is called when the operation is complete. You can pass the delegate a reference to the SqlCommand itself, making it easy to access the SqlCommand object and call the appropriate `End` method without having to use a global variable.

Example

The following Windows application demonstrates the use of the BeginExecuteNonQuery method, executing a Transact-SQL statement that includes a delay of a few seconds (emulating a long-running command).

This example demonstrates a number of important techniques, including calling a method that interacts with the form from a separate thread. In addition, this example demonstrates how you must block users from concurrently executing a command multiple times, and how you must ensure that the form does not close before the callback procedure is called.

To set up this example, create a new Windows application. Place a Button control and two Label controls on the form (accepting the default name for each control). Add the following code to the form's class, modifying the connection string as necessary for your environment.

```
' Add these to the top of the class:
Imports System
Imports System.Data
Imports System.Data.SqlClient

' Add this code to the form's class:

    ' You'll need this delegate in order to display text from a
    ' thread other than the form's thread. See the HandleCallback
    ' procedure for more information.
    ' This same delegate matches both the DisplayStatus
    ' and DisplayResults methods.
    Private Delegate Sub DisplayInfoDelegate(ByVal Text As String)

    ' This flag ensures that the user doesn't attempt
    ' to restart the command or close the form while the
    ' asynchronous command is executing.
    Private isExecuting As Boolean

    ' This example maintains the connection object
    ' externally, so that it's available for closing.
    Private connection As SqlConnection

    Private Function GetConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.

        ' If you have not included "Asynchronous Processing=true"
```

```
      If you have not included "Asynchronous Processing=true"
    ' in the connection string, the command will not be able
    ' to execute asynchronously.
    Return "Data Source=(local);Integrated Security=SSPI;" & _
      "Initial Catalog=AdventureWorks;" & _
      "Asynchronous Processing=true"
End Function

Private Sub DisplayStatus(ByVal Text As String)
    Me.Label1.Text = Text
End Sub

Private Sub DisplayResults(ByVal Text As String)
    Me.Label1.Text = Text
    DisplayStatus("Ready")
End Sub

Private Sub Form1_FormClosing(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.FormClosingEventArgs) _
    Handles Me.FormClosing
    If isExecuting Then
        MessageBox.Show(Me, "Can't close the form until " & _
         "the pending asynchronous command has completed. " & _
         "Please wait...")
        e.Cancel = True
    End If
End Sub

Private Sub Button1_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    If isExecuting Then
        MessageBox.Show(Me, _
            "Already executing. " & _
            "Please wait until the current query " & _
            "has completed.")
    Else
        Dim command As SqlCommand
        Try
            DisplayResults("")
            DisplayStatus("Connecting...")
            connection = New SqlConnection(GetConnectionString())
            ' To emulate a long-running query, wait for
            ' a few seconds before working with the data.
            ' This command doesn't do much, but that's the point--
            ' it doesn't change your data, in the long run.
            Dim commandText As String = _
                "WAITFOR DELAY '0:0:05';" & _
                "UPDATE Production.Product " & _
                "SET ReorderPoint = ReorderPoint + 1 " & _
                "WHERE ReorderPoint Is Not Null;" & _
                "UPDATE Production.Product " & _
                "SET ReorderPoint = ReorderPoint - 1 " & _
                "WHERE ReorderPoint Is Not Null"

            command = New SqlCommand(commandText, connection)
            connection.Open()

            DisplayStatus("Executing...")
            isExecuting = True
            ' Although it's not required that you pass the
            ' SqlCommand object as the second parameter in the
            ' BeginExecuteNonQuery call, doing so makes it easier
            ' to call EndExecuteNonQuery in the callback procedure.
            Dim callback As New _
                  AsyncCallback(AddressOf HandleCallback)

            ' Once the BeginExecuteNonQuery method is called,
            ' the code continues--and the user can interact with
            ' the form while the server executes the query
```

```
            ' the form--while the server executes the query.

            command.BeginExecuteNonQuery(callback, command)

        Catch ex As Exception
            isExecuting = False
            DisplayStatus($"Ready (last error: {ex.Message})")
            If connection IsNot Nothing Then
                connection.Close()
            End If
        End Try
    End If
End Sub

Private Sub HandleCallback(ByVal result As IAsyncResult)
    Try
        ' Retrieve the original command object, passed
        ' to this procedure in the AsyncState property
        ' of the IAsyncResult parameter.
        Dim command As SqlCommand = _
            CType(result.AsyncState, SqlCommand)
        Dim rowCount As Integer = _
            command.EndExecuteNonQuery(result)
        Dim rowText As String = " rows affected."
        If rowCount = 1 Then
            rowText = " row affected."
        End If
        rowText = rowCount & rowText

        ' You may not interact with the form and its contents
        ' from a different thread, and this callback procedure
        ' is all but guaranteed to be running from a different
        ' thread than the form. Therefore you cannot simply call
        ' code that displays the results, like this:
        ' DisplayResults(rowText)

        ' Instead, you must call the procedure from the form's
        ' thread. One simple way to accomplish this is to call
        ' the Invoke method of the form, which calls the delegate
        ' you supply from the form's thread.
        Dim del As New _
            DisplayInfoDelegate(AddressOf DisplayResults)
        Me.Invoke(del, rowText)

    Catch ex As Exception
        ' Because you're now running code in a separate thread,
        ' if you don't handle the exception here, none of your
        ' other code will catch the exception. Because none of
        ' your code is on the call stack in this thread, there's
        ' nothing higher up the stack to catch the exception if
        ' you don't handle it here. You can either log the
        ' exception or invoke a delegate (as in the non-error
        ' case in this example) to display the error on the form.
        ' In no case can you simply display the error without
        ' executing a delegate as in the Try block here.

        ' You can create the delegate instance as you
        ' invoke it, like this:
        Me.Invoke(New _
            DisplayInfoDelegate(AddressOf DisplayStatus), _
            $"Ready (last error: {ex.Message}")
    Finally
        isExecuting = False
        If connection IsNot Nothing Then
            connection.Close()
        End If
    End Try
End Sub
```

```
// Add these to the top of the class, if they're not already there:
using System;
using System.Data;
using System.Data.SqlClient;

// Hook up the form's Load event handler (you can double-click on
// the form's design surface in Visual Studio), and then add
// this code to the form's class:

// You'll need this delegate in order to display text from a thread
// other than the form's thread. See the HandleCallback
// procedure for more information.
// This same delegate matches both the DisplayStatus
// and DisplayResults methods.
private delegate void DisplayInfoDelegate(string Text);

// This flag ensures that the user doesn't attempt
// to restart the command or close the form while the
// asynchronous command is executing.
private bool isExecuting;

// This example maintains the connection object
// externally, so that it's available for closing.
private SqlConnection connection;

private static string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.

    // If you have not included "Asynchronous Processing=true" in the
    // connection string, the command will not be able
    // to execute asynchronously.
    return "Data Source=(local);Integrated Security=SSPI;" +
    "Initial Catalog=AdventureWorks; Asynchronous Processing=true";
}

private void DisplayStatus(string Text)
{
    this.label1.Text = Text;
}

private void DisplayResults(string Text)
{
    this.label1.Text = Text;
    DisplayStatus("Ready");
}

private void Form1_FormClosing(object sender, System.Windows.Forms.FormClosingEventArgs e)
{
    if (isExecuting)
    {
        MessageBox.Show(this, "Can't close the form until " +
        "the pending asynchronous command has completed. Please " +
        "wait...");
        e.Cancel = true;
    }
}

private void button1_Click(object sender, System.EventArgs e)
{
    if (isExecuting)
    {
        MessageBox.Show(this, "Already executing. Please wait until " +
        "the current query has completed.");
    }
    else
    {
```

```
        SqlCommand command = null;
        try
        {
            DisplayResults("");
            DisplayStatus("Connecting...");
            connection = new SqlConnection(GetConnectionString());
            // To emulate a long-running query, wait for
            // a few seconds before working with the data.
            // This command doesn't do much, but that's the point--
            // it doesn't change your data, in the long run.
            string commandText =
                "WAITFOR DELAY '0:0:05';" +
                "UPDATE Production.Product " +
                "SET ReorderPoint = ReorderPoint + 1 " +
                "WHERE ReorderPoint Is Not Null;" +
                "UPDATE Production.Product " +
                "SET ReorderPoint = ReorderPoint - 1 " +
                "WHERE ReorderPoint Is Not Null";

            command = new SqlCommand(commandText, connection);
            connection.Open();

            DisplayStatus("Executing...");
            isExecuting = true;
            // Although it's not required that you pass the
            // SqlCommand object as the second parameter in the
            // BeginExecuteNonQuery call, doing so makes it easier
            // to call EndExecuteNonQuery in the callback procedure.
            AsyncCallback callback = new AsyncCallback(HandleCallback);

            // Once the BeginExecuteNonQuery method is called,
            // the code continues--and the user can interact with
            // the form--while the server executes the query.
            command.BeginExecuteNonQuery(callback, command);

        }
        catch (Exception ex)
        {
            isExecuting = false;
            DisplayStatus($"Ready (last error: {ex.Message})");
            if (connection != null)
            {
                connection.Close();
            }
        }
    }
}

private void HandleCallback(IAsyncResult result)
{
    try
    {
        // Retrieve the original command object, passed
        // to this procedure in the AsyncState property
        // of the IAsyncResult parameter.
        SqlCommand command = (SqlCommand)result.AsyncState;
        int rowCount = command.EndExecuteNonQuery(result);
        string rowText = " rows affected.";
        if (rowCount == 1)
        {
            rowText = " row affected.";
        }
        rowText = rowCount + rowText;

        // You may not interact with the form and its contents
        // from a different thread, and this callback procedure
        // is all but guaranteed to be running from a different thread
        // than the form. Therefore you cannot simply call code that
        // displays the results, like this:
```

```
        // displays the results, like this:
        // DisplayResults(rowText)

        // Instead, you must call the procedure from the form's thread.
        // One simple way to accomplish this is to call the Invoke
        // method of the form, which calls the delegate you supply
        // from the form's thread.
        DisplayInfoDelegate del =
         new DisplayInfoDelegate(DisplayResults);
        this.Invoke(del, rowText);
    }
    catch (Exception ex)
    {
        // Because you're now running code in a separate thread,
        // if you don't handle the exception here, none of your other
        // code will catch the exception. Because none of your
        // code is on the call stack in this thread, there's nothing
        // higher up the stack to catch the exception if you don't
        // handle it here. You can either log the exception or
        // invoke a delegate (as in the non-error case in this
        // example) to display the error on the form. In no case
        // can you simply display the error without executing a
        // delegate as in the try block here.

        // You can create the delegate instance as you
        // invoke it, like this:
        this.Invoke(new DisplayInfoDelegate(DisplayStatus),
            $"Ready (last error: {ex.Message}");
    }
    finally
    {
        isExecuting = false;
        if (connection != null)
        {
            connection.Close();
        }
    }
}

private void Form1_Load(object sender, System.EventArgs e)
{
    this.button1.Click += new System.EventHandler(this.button1_Click);
    this.FormClosing += new System.Windows.Forms.
        FormClosingEventHandler(this.Form1_FormClosing);
}
```

See also

- Asynchronous Operations
- ADO.NET Overview

ASP.NET Applications Using Wait Handles

4/26/2022 • 12 minutes to read • Edit Online

The callback and polling models for handling asynchronous operations are useful when your application is processing only one asynchronous operation at a time. The Wait models provide a more flexible way of processing multiple asynchronous operations. There are two Wait models, named for the WaitHandle methods used to implement them: the Wait (Any) model and the Wait (All) model.

To use either Wait model, you need to use the AsyncWaitHandle property of the IAsyncResult object returned by the BeginExecuteNonQuery, BeginExecuteReader, or BeginExecuteXmlReader methods. The WaitAny and WaitAll methods both require you to send the WaitHandle objects as an argument, grouped together in an array.

Both Wait methods monitor the asynchronous operations, waiting for completion. The WaitAny method waits for any of the operations to complete or time out. Once you know a particular operation is complete, you can process its results and then continue waiting for the next operation to complete or time out. The WaitAll method waits for all of the processes in the array of WaitHandle instances to complete or time out before continuing.

The Wait models' benefit is most striking when you need to run multiple operations of some length on different servers, or when your server is powerful enough to process all the queries at the same time. In the examples presented here, three queries emulate long processes by adding WAITFOR commands of varying lengths to inconsequential SELECT queries.

Example: Wait (Any) Model

The following example illustrates the Wait (Any) model. Once three asynchronous processes are started, the WaitAny method is called to wait for the completion of any one of them. As each process completes, the EndExecuteReader method is called and the resulting SqlDataReader object is read. At this point, a real-world application would likely use the SqlDataReader to populate a portion of the page. In this simple example, the time the process completed is added to a text box corresponding to the process. Taken together, the times in the text boxes illustrate the point: Code is executed each time a process completes.

To set up this example, create a new ASP.NET Web Site project. Place a Button control and four TextBox controls on the page (accepting the default name for each control).

Add the following code to the form's class, modifying the connection string as necessary for your environment.

```
' Add these to the top of the class
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Threading

' Add this code to the page's class:
    Private Function GetConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.

        ' If you have not included "Asynchronous Processing=true"
        ' in the connection string, the command will not be able
        ' to execute asynchronously.
        Return "Data Source=(local);Integrated Security=SSPI;" & _
          "Initial Catalog=AdventureWorks;" & _
          "Asynchronous Processing=true"
    End Function

    Sub Button1_Click(
```

```
Sub Button1_Click( _
 ByVal sender As Object, ByVal e As System.EventArgs)

    ' In a real-world application, you might be connecting to
    '  three different servers or databases. For the example,
    '  we connect to only one.
    Dim connection1 As New SqlConnection(GetConnectionString())
    Dim connection2 As New SqlConnection(GetConnectionString())
    Dim connection3 As New SqlConnection(GetConnectionString())

    ' To keep the example simple, all three asynchronous
    ' processes select a row from the same table. WAITFOR
    ' commands are used to emulate long-running processes
    ' that complete after different periods of time.
    Dim commandText1 As String = _
        "WAITFOR DELAY '0:0:01';" & _
        "SELECT * FROM Production.Product " & _
        "WHERE ProductNumber = 'BL-2036'"

    Dim commandText2 As String = _
        "WAITFOR DELAY '0:0:05';" & _
        "SELECT * FROM Production.Product " & _
        "WHERE ProductNumber = 'BL-2036'"

    Dim commandText3 As String = _
        "WAITFOR DELAY '0:0:10';" & _
        "SELECT * FROM Production.Product " & _
        "WHERE ProductNumber = 'BL-2036'"

    Dim waitHandles(2) As WaitHandle
    Try
        ' For each process, open a connection and begin execution.
        ' Use the IAsyncResult object returned by
        ' BeginExecuteReader to add a WaitHandle for the process
        ' to the array.
        connection1.Open()
        Dim command1 As New SqlCommand(commandText1, connection1)
        Dim result1 As IAsyncResult = _
         command1.BeginExecuteReader()
        waitHandles(0) = result1.AsyncWaitHandle

        connection2.Open()
        Dim command2 As New SqlCommand(commandText2, connection2)
        Dim result2 As IAsyncResult = _
         command2.BeginExecuteReader()
        waitHandles(1) = result2.AsyncWaitHandle

        connection3.Open()
        Dim command3 As New SqlCommand(commandText3, connection3)
        Dim result3 As IAsyncResult = _
         command3.BeginExecuteReader()
        waitHandles(2) = result3.AsyncWaitHandle

        Dim index As Integer
        For countWaits As Integer = 1 To 3
            ' WaitAny waits for any of the processes to complete.
            ' The return value is either the index of the
            ' array element whose process just completed, or
            ' the WaitTimeout value.
            index = WaitHandle.WaitAny(waitHandles, 60000, False)
            ' This example doesn't actually do anything with the
            ' data returned by the processes, but the code opens
            ' readers for each just to demonstrate the concept.
            ' Instead of using the returned data to fill the
            ' controls on the page, the example adds the time
            ' the process was completed to the corresponding
            ' text box.
            Select Case index
                Case 0
                    Dim reader1 As SqlDataReader
```

```
                        Dim reader1 As SqlDataReader
                        reader1 = command1.EndExecuteReader(result1)
                        If reader1.Read Then
                            TextBox1.Text = _
                             "Completed " & _
                             System.DateTime.Now.ToLongTimeString()
                        End If
                        reader1.Close()

                    Case 1
                        Dim reader2 As SqlDataReader
                        reader2 = command2.EndExecuteReader(result2)
                        If reader2.Read Then
                            TextBox2.Text = _
                             "Completed " & _
                             System.DateTime.Now.ToLongTimeString()
                        End If
                        reader2.Close()
                    Case 2
                        Dim reader3 As SqlDataReader
                        reader3 = command3.EndExecuteReader(result3)
                        If reader3.Read Then
                            TextBox3.Text = _
                             "Completed " & _
                             System.DateTime.Now.ToLongTimeString()
                        End If
                        reader3.Close()
                    Case WaitHandle.WaitTimeout
                        Throw New Exception("Timeout")
                End Select

            Next
        Catch ex As Exception
            TextBox4.Text = ex.ToString
        End Try
        connection1.Close()
        connection2.Close()
        connection3.Close()

    End Sub
```

```
// Add the following using statements, if they are not already there.
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Threading;
using System.Data.SqlClient;

// Add this code to the page's class
string GetConnectionString()
     //  To avoid storing the connection string in your code,
     //  you can retrieve it from a configuration file.
     //  If you have not included "Asynchronous Processing=true"
     //  in the connection string, the command will not be able
     //  to execute asynchronously.
{
     return "Data Source=(local);Integrated Security=SSPI;" +
          "Initial Catalog=AdventureWorks;" +
          "Asynchronous Processing=true";
}
void Button1_Click(object sender, System.EventArgs e)
{
```

```
{
    //  In a real-world application, you might be connecting to
    //   three different servers or databases. For the example,
    //   we connect to only one.

    SqlConnection connection1 =
         new SqlConnection(GetConnectionString());
    SqlConnection connection2 =
         new SqlConnection(GetConnectionString());
    SqlConnection connection3 =
         new SqlConnection(GetConnectionString());
    //  To keep the example simple, all three asynchronous
    //  processes select a row from the same table. WAITFOR
    //  commands are used to emulate long-running processes
    //  that complete after different periods of time.

    string commandText1 = "WAITFOR DELAY '0:0:01';" +
         "SELECT * FROM Production.Product " +
         "WHERE ProductNumber = 'BL-2036'";
    string commandText2 = "WAITFOR DELAY '0:0:05';" +
         "SELECT * FROM Production.Product " +
         "WHERE ProductNumber = 'BL-2036'";
    string commandText3 = "WAITFOR DELAY '0:0:10';" +
         "SELECT * FROM Production.Product " +
         "WHERE ProductNumber = 'BL-2036'";
    try
         //  For each process, open a connection and begin
         //  execution. Use the IAsyncResult object returned by
         //  BeginExecuteReader to add a WaitHandle for the
         //  process to the array.
    {
         connection1.Open();
         SqlCommand command1 =
              new SqlCommand(commandText1, connection1);
         IAsyncResult result1 = command1.BeginExecuteReader();
         WaitHandle waitHandle1 = result1.AsyncWaitHandle;

         connection2.Open();
         SqlCommand command2 =
              new SqlCommand(commandText2, connection2);
         IAsyncResult result2 = command2.BeginExecuteReader();
         WaitHandle waitHandle2 = result2.AsyncWaitHandle;

         connection3.Open();
         SqlCommand command3 =
              new SqlCommand(commandText3, connection3);
         IAsyncResult result3 = command3.BeginExecuteReader();
         WaitHandle waitHandle3 = result3.AsyncWaitHandle;

         WaitHandle[] waitHandles = {
              waitHandle1, waitHandle2, waitHandle3
         };

         int index;
         for (int countWaits = 0; countWaits <= 2; countWaits++)
         {
              //  WaitAny waits for any of the processes to
              //  complete. The return value is either the index
              //  of the array element whose process just
              //  completed, or the WaitTimeout value.

              index = WaitHandle.WaitAny(waitHandles,
                   60000, false);
              //  This example doesn't actually do anything with
              //  the data returned by the processes, but the
              //  code opens readers for each just to demonstrate
              //  the concept.
              //  Instead of using the returned data to fill the
              //  controls on the page, the example adds the time
```

```
                //  the process was completed to the corresponding
                //  text box.

                switch (index)
                {
                     case 0:
                          SqlDataReader reader1;
                          reader1 =
                               command1.EndExecuteReader(result1);
                          if (reader1.Read())
                          {
                            TextBox1.Text =
                            "Completed " +
                            System.DateTime.Now.ToLongTimeString();
                          }
                          reader1.Close();
                          break;
                     case 1:
                          SqlDataReader reader2;
                          reader2 =
                               command2.EndExecuteReader(result2);
                          if (reader2.Read())
                          {
                            TextBox2.Text =
                            "Completed " +
                            System.DateTime.Now.ToLongTimeString();
                          }
                          reader2.Close();
                          break;
                     case 2:
                          SqlDataReader reader3;
                          reader3 =
                               command3.EndExecuteReader(result3);
                          if (reader3.Read())
                          {
                            TextBox3.Text =
                            "Completed " +
                            System.DateTime.Now.ToLongTimeString();
                          }
                          reader3.Close();
                          break;
                     case WaitHandle.WaitTimeout:
                          throw new Exception("Timeout");
                          break;
                }
           }
      }
      catch (Exception ex)
      {
           TextBox4.Text = ex.ToString();
      }
      connection1.Close();
      connection2.Close();
      connection3.Close();
}
```

Example: Wait (All) Model

The following example illustrates the Wait (All) model. Once three asynchronous processes are started, the WaitAll method is called to wait for the processes to complete or time out.

Like the example of the Wait (Any) model, the time the process completed is added to a text box corresponding to the process. Again, the times in the text boxes illustrate the point: Code following the WaitAny method is executed only after all processes are complete.

To set up this example, create a new ASP.NET Web Site project. Place a Button control and four TextBox controls

on the page (accepting the default name for each control).

Add the following code to the form's class, modifying the connection string as necessary for your environment.

```
' Add these to the top of the class
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Threading

' Add this code to the page's class:
    Private Function GetConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.

        ' If you have not included "Asynchronous Processing=true"
        ' in the connection string, the command will not be able
        ' to execute asynchronously.
        Return "Data Source=(local);Integrated Security=SSPI;" & _
          "Initial Catalog=AdventureWorks;" & _
          "Asynchronous Processing=true"
    End Function
    Sub Button1_Click( _
     ByVal sender As Object, ByVal e As System.EventArgs)

        ' In a real-world application, you might be connecting to
        '  three different servers or databases. For the example,
        '  we connect to only one.
        Dim connection1 As New SqlConnection(GetConnectionString())
        Dim connection2 As New SqlConnection(GetConnectionString())
        Dim connection3 As New SqlConnection(GetConnectionString())

        ' To keep the example simple, all three asynchronous
        ' processes select a row from the same table. WAITFOR
        ' commands are used to emulate long-running processes
        ' that complete after different periods of time.
        Dim commandText1 As String = _
         "UPDATE Production.Product " & _
         "SET ReorderPoint = ReorderPoint + 1 " & _
         "WHERE ReorderPoint Is Not Null;" & _
         "WAITFOR DELAY '0:0:01';" & _
         "UPDATE Production.Product " & _
         "SET ReorderPoint = ReorderPoint - 1 " & _
         "WHERE ReorderPoint Is Not Null"

        Dim commandText2 As String = _
         "UPDATE Production.Product " & _
         "SET ReorderPoint = ReorderPoint + 1 " & _
         "WHERE ReorderPoint Is Not Null;" & _
         "WAITFOR DELAY '0:0:05';" & _
         "UPDATE Production.Product " & _
         "SET ReorderPoint = ReorderPoint - 1 " & _
         "WHERE ReorderPoint Is Not Null"

        Dim commandText3 As String = _
         "UPDATE Production.Product " & _
         "SET ReorderPoint = ReorderPoint + 1 " & _
         "WHERE ReorderPoint Is Not Null;" & _
         "WAITFOR DELAY '0:0:10';" & _
         "UPDATE Production.Product " & _
         "SET ReorderPoint = ReorderPoint - 1 " & _
         "WHERE ReorderPoint Is Not Null"

        Dim waitHandles(2) As WaitHandle

        Try
            ' For each process, open a connection and begin execution.
            ' Use the IAsyncResult object returned by
```

```
            ' BeginExecuteReader to add a WaitHandle for the process
            ' to the array.
            connection1.Open()
            Dim command1 As New SqlCommand(commandText1, connection1)
            Dim result1 As IAsyncResult = _
             command1.BeginExecuteNonQuery()
            waitHandles(0) = result1.AsyncWaitHandle

            connection2.Open()
            Dim command2 As New SqlCommand(commandText2, connection2)
            Dim result2 As IAsyncResult = _
             command2.BeginExecuteNonQuery()
            waitHandles(1) = result2.AsyncWaitHandle

            connection3.Open()
            Dim command3 As New SqlCommand(commandText3, connection3)
            Dim result3 As IAsyncResult = _
             command3.BeginExecuteNonQuery()
            waitHandles(2) = result3.AsyncWaitHandle

            ' WaitAll waits for all of the processes to complete.
            ' The return value is True if all processes completed,
            ' False if any process timed out.
            Dim result As Boolean = _
             WaitHandle.WaitAll(waitHandles, 60000, False)
            If result Then
                Dim rowCount1 As Long = _
                 command1.EndExecuteNonQuery(result1)
                TextBox1.Text = _
                 "Completed " & _
                 System.DateTime.Now.ToLongTimeString()

                Dim rowCount2 As Long = _
                 command2.EndExecuteNonQuery(result2)
                TextBox2.Text = _
                 "Completed " & _
                 System.DateTime.Now.ToLongTimeString()

                Dim rowCount3 As Long = _
                 command3.EndExecuteNonQuery(result3)
                TextBox3.Text = _
                 "Completed " & _
                 System.DateTime.Now.ToLongTimeString()
            Else
                Throw New Exception("Timeout")
            End If
        Catch ex As Exception
            TextBox4.Text = ex.ToString
        End Try
        connection1.Close()
        connection2.Close()
        connection3.Close()

    End Sub
```

```
// Add the following using statements, if they are not already there.
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
using System.Threading;
using System.Data.SqlClient;
```

```
// Add this code to the page's class
string GetConnectionString()
    //  To avoid storing the connection string in your code,
    //  you can retrieve it from a configuration file.
    //  If you have not included "Asynchronous Processing=true"
    //  in the connection string, the command will not be able
    //  to execute asynchronously.
{
    return "Data Source=(local);Integrated Security=SSPI;" +
        "Initial Catalog=AdventureWorks;" +
        "Asynchronous Processing=true";
}
void Button1_Click(object sender, System.EventArgs e)
{
    //  In a real-world application, you might be connecting to
    //   three different servers or databases. For the example,
    //   we connect to only one.

    SqlConnection connection1 =
        new SqlConnection(GetConnectionString());
    SqlConnection connection2 =
        new SqlConnection(GetConnectionString());
    SqlConnection connection3 =
        new SqlConnection(GetConnectionString());
    //  To keep the example simple, all three asynchronous
    //  processes execute UPDATE queries that result in
      //  no change to the data. WAITFOR
    //  commands are used to emulate long-running processes
    //  that complete after different periods of time.

    string commandText1 =
        "UPDATE Production.Product " +
        "SET ReorderPoint = ReorderPoint + 1 " +
        "WHERE ReorderPoint Is Not Null;" +
        "WAITFOR DELAY '0:0:01';" +
        "UPDATE Production.Product " +
        "SET ReorderPoint = ReorderPoint - 1 " +
        "WHERE ReorderPoint Is Not Null";

    string commandText2 =
      "UPDATE Production.Product " +
      "SET ReorderPoint = ReorderPoint + 1 " +
      "WHERE ReorderPoint Is Not Null;" +
      "WAITFOR DELAY '0:0:05';" +
      "UPDATE Production.Product " +
      "SET ReorderPoint = ReorderPoint - 1 " +
      "WHERE ReorderPoint Is Not Null";

    string commandText3 =
       "UPDATE Production.Product " +
       "SET ReorderPoint = ReorderPoint + 1 " +
       "WHERE ReorderPoint Is Not Null;" +
       "WAITFOR DELAY '0:0:10';" +
       "UPDATE Production.Product " +
       "SET ReorderPoint = ReorderPoint - 1 " +
       "WHERE ReorderPoint Is Not Null";
    try
        //  For each process, open a connection and begin
        //  execution. Use the IAsyncResult object returned by
        //  BeginExecuteReader to add a WaitHandle for the
        //  process to the array.
    {
        connection1.Open();
        SqlCommand command1 =
            new SqlCommand(commandText1, connection1);
        IAsyncResult result1 = command1.BeginExecuteNonQuery();
        WaitHandle waitHandle1 = result1.AsyncWaitHandle;
        connection2.Open();
```

```
        connection2.Open();

        SqlCommand command2 =
            new SqlCommand(commandText2, connection2);
        IAsyncResult result2 = command2.BeginExecuteNonQuery();
        WaitHandle waitHandle2 = result2.AsyncWaitHandle;
        connection3.Open();

        SqlCommand command3 =
            new SqlCommand(commandText3, connection3);
        IAsyncResult result3 = command3.BeginExecuteNonQuery();
        WaitHandle waitHandle3 = result3.AsyncWaitHandle;

        WaitHandle[] waitHandles = {
            waitHandle1, waitHandle2, waitHandle3
        };

        bool result;
        //  WaitAll waits for all of the processes to
        //  complete. The return value is True if the processes
        //  all completed successfully, False if any process
        //  timed out.

        result = WaitHandle.WaitAll(waitHandles, 60000, false);
        if(result)
        {
            long rowCount1 =
                command1.EndExecuteNonQuery(result1);
            TextBox1.Text = "Completed " +
                System.DateTime.Now.ToLongTimeString();
            long rowCount2 =
                command2.EndExecuteNonQuery(result2);
            TextBox2.Text = "Completed " +
                System.DateTime.Now.ToLongTimeString();

            long rowCount3 =
                command3.EndExecuteNonQuery(result3);
            TextBox3.Text = "Completed " +
                System.DateTime.Now.ToLongTimeString();
        }
        else
        {
            throw new Exception("Timeout");
        }
    }

    catch (Exception ex)
    {
        TextBox4.Text = ex.ToString();
    }
    connection1.Close();
    connection2.Close();
    connection3.Close();
}
```

See also

- Asynchronous Operations
- ADO.NET Overview

Polling in Console Applications

4/26/2022 • 3 minutes to read • Edit Online

Asynchronous operations in ADO.NET allow you to initiate time-consuming database operations on one thread while performing other tasks on another thread. In most scenarios, however, you will eventually reach a point where your application should not continue until the database operation is complete. For such cases, it is useful to poll the asynchronous operation to determine whether the operation has completed or not.

You can use the IsCompleted property to find out whether or not the operation has completed.

Example

The following console application updates data within the **AdventureWorks** sample database, doing its work asynchronously. In order to emulate a long-running process, this example inserts a WAITFOR statement in the command text. Normally, you would not try to make your commands run slower, but doing so in this case makes it easier to demonstrate asynchronous behavior.

```
Imports System
Imports System.Data.SqlClient

Module Module1

    Sub Main()
        ' The WAITFOR statement simply adds enough time to prove the
        ' asynchronous nature of the command.
        Dim commandText As String = _
         "UPDATE Production.Product " & _
         "SET ReorderPoint = ReorderPoint + 1 " & _
         "WHERE ReorderPoint Is Not Null;" & _
         "WAITFOR DELAY '0:0:3';" & _
         "UPDATE Production.Product " & _
         "SET ReorderPoint = ReorderPoint - 1 " & _
         "WHERE ReorderPoint Is Not Null"

        RunCommandAsynchronously(commandText, GetConnectionString())

        Console.WriteLine("Press Enter to continue.")
        Console.ReadLine()
    End Sub

    Private Sub RunCommandAsynchronously( _
     ByVal commandText As String, ByVal connectionString As String)

        ' Given command text and connection string, asynchronously
        ' execute the specified command against the connection. For
        ' this example, the code displays an indicator as it's working,
        ' verifying the asynchronous behavior.
        Using connection As New SqlConnection(connectionString)
            Try
                Dim count As Integer = 0
                Dim command As New SqlCommand(commandText, connection)
                connection.Open()
                Dim result As IAsyncResult = _
                 command.BeginExecuteNonQuery()
                While Not result.IsCompleted
                    Console.WriteLine("Waiting ({0})", count)
                    ' Wait for 1/10 second, so the counter
                    ' doesn't consume all available resources
                    ' on the main thread.
```

```
                    Threading.Thread.Sleep(100)
                    count += 1
                End While
                Console.WriteLine( _
                 "Command complete. Affected {0} rows.", _
                 command.EndExecuteNonQuery(result))
            Catch ex As SqlException
                Console.WriteLine("Error ({0}): {1}", _
                 ex.Number, ex.Message)
            Catch ex As InvalidOperationException
                Console.WriteLine("Error: {0}", ex.Message)
            Catch ex As Exception
                ' You might want to pass these errors
                ' back out to the caller.
                Console.WriteLine("Error: {0}", ex.Message)
            End Try
        End Using
    End Sub

    Private Function GetConnectionString() As String
        ' To avoid storing the connection string in your code,
        ' you can retrieve it from a configuration file.

        ' If you have not included "Asynchronous Processing=true"
        ' in the connection string, the command will not be able
        ' to execute asynchronously.
        Return "Data Source=(local);Integrated Security=SSPI;" & _
          "Initial Catalog=AdventureWorks; " & _
          "Asynchronous Processing=true"
    End Function
End Module
```

```
using System;
using System.Data;
using System.Data.SqlClient;

class Class1
{
    [STAThread]
    static void Main()
    {
        // The WAITFOR statement simply adds enough time to
        // prove the asynchronous nature of the command.

        string commandText =
          "UPDATE Production.Product SET ReorderPoint = " +
          "ReorderPoint + 1 " +
          "WHERE ReorderPoint Is Not Null;" +
          "WAITFOR DELAY '0:0:3';" +
          "UPDATE Production.Product SET ReorderPoint = " +
          "ReorderPoint - 1 " +
          "WHERE ReorderPoint Is Not Null";

        RunCommandAsynchronously(
            commandText, GetConnectionString());

        Console.WriteLine("Press Enter to continue.");
        Console.ReadLine();
    }

    private static void RunCommandAsynchronously(
      string commandText, string connectionString)
    {
        // Given command text and connection string, asynchronously
        // execute the specified command against the connection.
        // For this example, the code displays an indicator as it's
        // working, verifying the asynchronous behavior.
```

```
        using (SqlConnection connection =
          new SqlConnection(connectionString))
        {
            try
            {
                int count = 0;
                SqlCommand command =
                    new SqlCommand(commandText, connection);
                connection.Open();

                IAsyncResult result =
                    command.BeginExecuteNonQuery();
                while (!result.IsCompleted)
                {
                    Console.WriteLine(
                                    "Waiting ({0})", count++);
                    // Wait for 1/10 second, so the counter
                    // doesn't consume all available
                    // resources on the main thread.
                    System.Threading.Thread.Sleep(100);
                }
                Console.WriteLine(
                    "Command complete. Affected {0} rows.",
                command.EndExecuteNonQuery(result));
            }
            catch (SqlException ex)
            {
                Console.WriteLine("Error ({0}): {1}",
                    ex.Number, ex.Message);
            }
            catch (InvalidOperationException ex)
            {
                Console.WriteLine("Error: {0}", ex.Message);
            }
            catch (Exception ex)
            {
                // You might want to pass these errors
                // back out to the caller.
                Console.WriteLine("Error: {0}", ex.Message);
            }
        }
    }

    private static string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.

        // If you have not included "Asynchronous Processing=true"
        // in the connection string, the command will not be able
        // to execute asynchronously.
        return "Data Source=(local);Integrated Security=SSPI;" +
        "Initial Catalog=AdventureWorks; " +
        "Asynchronous Processing=true";
    }
}
```

See also

- Asynchronous Operations
- ADO.NET Overview

Table-Valued Parameters

4/26/2022 • 8 minutes to read • Edit Online

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 statement to include the 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 submitted in batches that contain multiple statements, each statement is 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 columns:

```
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.

```
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":

```
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.

```
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.

```
// 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";
```

```
' Configure the command and parameter.
Dim insertCommand As New SqlCommand(sqlInsert, connection)
Dim tvpParam As SqlParameter = _
   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:

```
// 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;
```

```
' Configure the SqlCommand and table-valued parameter.
Dim insertCommand As New SqlCommand("usp_InsertCategories", connection)
insertCommand.CommandType = CommandType.StoredProcedure
Dim tvpParam As SqlParameter = _
  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.

```
// Assumes connection is an open SqlConnection object.
using (connection)
{
  // Create a DataTable with the modified rows.
  DataTable addedCategories = CategoriesDataTable.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", addedCategories);
  tvpParam.SqlDbType = SqlDbType.Structured;

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

```
' Assumes connection is an open SqlConnection object.
Using connection
   '  Create a DataTable with the modified rows.
   Dim addedCategories As DataTable = _
     CategoriesDataTable.GetChanges(DataRowState.Added)

  ' Configure the SqlCommand and SqlParameter.
   Dim insertCommand As New SqlCommand( _
     "usp_InsertCategories", connection)
   insertCommand.CommandType = CommandType.StoredProcedure
   Dim tvpParam As SqlParameter = _
     insertCommand.Parameters.AddWithValue( _
     "@tvpNewCategories", addedCategories)
   tvpParam.SqlDbType = SqlDbType.Structured

   '  Execute the command.
   insertCommand.ExecuteNonQuery()
End Using
```

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.

```
// Assumes connection is an open SqlConnection.
using (connection)
{
  // Create a DataTable with the modified rows.
  DataTable addedCategories = CategoriesDataTable.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", addedCategories);
  tvpParam.SqlDbType = SqlDbType.Structured;
  tvpParam.TypeName = "dbo.CategoryTableType";

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

```
' Assumes connection is an open SqlConnection.
Using connection
  ' Create a DataTable with the modified rows.
  Dim addedCategories As DataTable = _
    CategoriesDataTable.GetChanges(DataRowState.Added)

  ' Define the INSERT-SELECT statement.
  Dim sqlInsert As String = _
  "INSERT INTO dbo.Categories (CategoryID, CategoryName)" _
  & " SELECT nc.CategoryID, nc.CategoryName" _
  & " FROM @tvpNewCategories AS nc;"

  ' Configure the command and parameter.
  Dim insertCommand As New SqlCommand(sqlInsert, connection)
  Dim tvpParam As SqlParameter = _
     insertCommand.Parameters.AddWithValue( _
    "@tvpNewCategories", addedCategories)
  tvpParam.SqlDbType = SqlDbType.Structured
  tvpParam.TypeName = "dbo.CategoryTableType"

  ' Execute the query
  insertCommand.ExecuteNonQuery()
End Using
```

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.

```
// 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();
```

```
' Assumes connection is an open SqlConnection.
' Retrieve data from Oracle.
Dim selectCommand As New OracleCommand( _
  "Select CategoryID, CategoryName FROM Categories;", _
  oracleConnection)
Dim oracleReader As OracleDataReader = _
  selectCommand.ExecuteReader(CommandBehavior.CloseConnection)

' Configure SqlCommand and table-valued parameter.
Dim insertCommand As New SqlCommand("usp_InsertCategories", connection)
insertCommand.CommandType = CommandType.StoredProcedure
Dim tvpParam As SqlParameter = _
  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

SQL Server Features and ADO.NET

4/26/2022 • 2 minutes to read • Edit Online

The articles in this section discuss features in SQL Server that are targeted at developing database applications using ADO.NET.

For more information, see Development (Database Engine).

In This Section

Enumerating Instances of SQL Server (ADO.NET)
Describes how to enumerate active instances of SQL Server.

Provider Statistics for SQL Server
Describes support for obtaining SQL Server run-time statistics.

SQL Server Express User Instances
Describes support for SQL Server Express user instances.

Database Mirroring in SQL Server
Describes database mirroring functionality.

SQL Server Common Language Runtime Integration
Describes how data can be accessed from within a common language runtime (CLR) database object in SQL Server.

Query Notifications in SQL Server
Describes how .NET Framework applications can request notification from SQL Server when data has changed.

Snapshot Isolation in SQL Server
Describes support for snapshot isolation, a row versioning mechanism designed to reduce blocking in transactional applications.

SqlClient Support for High Availability, Disaster Recovery
Describes SqlClient support for high-availability, disaster recovery (AlwaysOn) availability groups.

SqlClient Support for LocalDB
Describes SqlClient support for LocalDB databases.

See also

- SQL Server Data Operations in ADO.NET
- Retrieving and Modifying Data in ADO.NET
- LINQ to SQL
- SQL Server and ADO.NET
- ADO.NET Overview

Enumerating Instances of SQL Server (ADO.NET)

4/26/2022 • 3 minutes to read • Edit Online

SQL Server permits applications to find SQL Server instances within the current network. The SqlDataSourceEnumerator class exposes this information to the application developer, providing a DataTable containing information about all the visible servers. This returned table contains a list of server instances available on the network that matches the list provided when a user attempts to create a new connection, and expands the drop-down list containing all the available servers on the **Connection Properties** dialog box. The results displayed are not always complete.

> **NOTE**
> As with most Windows services, it is best to run the SQL Browser service with the least possible privileges.

Retrieving an Enumerator Instance

In order to retrieve the table containing information about the available SQL Server instances, you must first retrieve an enumerator, using the shared/static Instance property:

```
Dim instance As System.Data.Sql.SqlDataSourceEnumerator = _
   System.Data.Sql.SqlDataSourceEnumerator.Instance
```

```
System.Data.Sql.SqlDataSourceEnumerator instance =
   System.Data.Sql.SqlDataSourceEnumerator.Instance
```

Once you have retrieved the static instance, you can call the GetDataSources method, which returns a DataTable containing information about the available servers:

```
Dim dataTable As System.Data.DataTable = instance.GetDataSources()
```

```
System.Data.DataTable dataTable = instance.GetDataSources();
```

The table returned from the method call contains the following columns, all of which contain `string` values:

COLUMN	DESCRIPTION
ServerName	Name of the server.
InstanceName	Name of the server instance. Blank if the server is running as the default instance.
IsClustered	Indicates whether the server is part of a cluster.

COLUMN	DESCRIPTION
Version	Version of the server. For example: - 9.00.x (SQL Server 2005) - 10.0.xx (SQL Server 2008) - 10.50.x (SQL Server 2008 R2) - 11.0.xx (SQL Server 2012)

Enumeration Limitations

All of the available servers may or may not be listed. The list can vary depending on factors such as timeouts and network traffic. This can cause the list to be different on two consecutive calls. Only servers on the same network will be listed. Broadcast packets typically won't traverse routers, which is why you may not see a server listed, but it will be stable across calls.

Listed servers may or may not have additional information such as `IsClustered` and version. This is dependent on how the list was obtained. Servers listed through the SQL Server browser service will have more details than those found through the Windows infrastructure, which will list only the name.

> **NOTE**
> Server enumeration is only available when running in full-trust. Assemblies running in a partially-trusted environment will not be able to use it, even if they have the SqlClientPermission Code Access Security (CAS) permission.

SQL Server provides information for the SqlDataSourceEnumerator through the use of an external Windows service named SQL Browser. This service is enabled by default, but administrators may turn it off or disable it, making the server instance invisible to this class.

Example

The following console application retrieves information about all of the visible SQL Server instances and displays the information in the console window.

```
Imports System.Data.Sql

Module Module1
  Sub Main()
    ' Retrieve the enumerator instance and then the data.
    Dim instance As SqlDataSourceEnumerator = _
     SqlDataSourceEnumerator.Instance
    Dim table As System.Data.DataTable = instance.GetDataSources()

    ' Display the contents of the table.
    DisplayData(table)

    Console.WriteLine("Press any key to continue.")
    Console.ReadKey()
  End Sub

  Private Sub DisplayData(ByVal table As DataTable)
    For Each row As DataRow In table.Rows
      For Each col As DataColumn In table.Columns
        Console.WriteLine("{0} = {1}", col.ColumnName, row(col))
      Next
      Console.WriteLine("============================")
    Next
  End Sub
End Module
```

```
using System.Data.Sql;

class Program
{
  static void Main()
  {
    // Retrieve the enumerator instance and then the data.
    SqlDataSourceEnumerator instance =
      SqlDataSourceEnumerator.Instance;
    System.Data.DataTable table = instance.GetDataSources();

    // Display the contents of the table.
    DisplayData(table);

    Console.WriteLine("Press any key to continue.");
    Console.ReadKey();
  }

  private static void DisplayData(System.Data.DataTable table)
  {
    foreach (System.Data.DataRow row in table.Rows)
    {
      foreach (System.Data.DataColumn col in table.Columns)
      {
        Console.WriteLine("{0} = {1}", col.ColumnName, row[col]);
      }
      Console.WriteLine("============================");
    }
  }
}
```

See also

- SQL Server and ADO.NET
- ADO.NET Overview

Provider Statistics for SQL Server

4/26/2022 • 9 minutes to read • Edit Online

Starting with the .NET Framework version 2.0, the .NET Framework Data Provider for SQL Server supports run-time statistics. You must enable statistics by setting the StatisticsEnabled property of the SqlConnection object to `True` after you have a valid connection object created. After statistics are enabled, you can review them as a "snapshot in time" by retrieving an IDictionary reference via the RetrieveStatistics method of the SqlConnection object. You enumerate through the list as a set of name/value pair dictionary entries. These name/value pairs are unordered. At any time, you can call the ResetStatistics method of the SqlConnection object to reset the counters. If statistic gathering has not been enabled, an exception is not generated. In addition, if RetrieveStatistics is called without StatisticsEnabled having been called first, the values retrieved are the initial values for each entry. If you enable statistics, run your application for a while, and then disable statistics, the values retrieved will reflect the values collected up to the point where statistics were disabled. All statistical values gathered are on a per-connection basis.

Statistical Values Available

Currently there are 18 different items available from the Microsoft SQL Server provider. The number of items available can be accessed via the **Count** property of the IDictionary interface reference returned by RetrieveStatistics. All of the counters for provider statistics use the common language runtime Int64 type (**long** in C# and Visual Basic), which is 64 bits wide. The maximum value of the **int64** data type, as defined by the **int64.MaxValue** field, is ((2^63)-1)). When the values for the counters reach this maximum value, they should no longer be considered accurate. This means that **int64.MaxValue**-1((2^63)-2) is effectively the greatest valid value for any statistic.

> **NOTE**
> A dictionary is used for returning provider statistics because the number, names and order of the returned statistics may change in the future. Applications should not rely on a specific value being found in the dictionary, but should instead check whether the value is there and branch accordingly.

The following table describes the current statistical values available. Note that the key names for the individual values are not localized across regional versions of the Microsoft .NET Framework.

KEY NAME	DESCRIPTION
`BuffersReceived`	Returns the number of tabular data stream (TDS) packets received by the provider from SQL Server after the application has started using the provider and has enabled statistics.
`BuffersSent`	Returns the number of TDS packets sent to SQL Server by the provider after statistics have been enabled. Large commands can require multiple buffers. For example, if a large command is sent to the server and it requires six packets, `ServerRoundtrips` is incremented by one and `BuffersSent` is incremented by six.

KEY NAME	DESCRIPTION
`BytesReceived`	Returns the number of bytes of data in the TDS packets received by the provider from SQL Server once the application has started using the provider and has enabled statistics.
`BytesSent`	Returns the number of bytes of data sent to SQL Server in TDS packets after the application has started using the provider and has enabled statistics.
`ConnectionTime`	The amount of time (in milliseconds) that the connection has been opened after statistics have been enabled (total connection time if statistics were enabled before opening the connection).
`CursorOpens`	Returns the number of times a cursor was open through the connection once the application has started using the provider and has enabled statistics. Note that read-only/forward-only results returned by SELECT statements are not considered cursors and thus do not affect this counter.
`ExecutionTime`	Returns the cumulative amount of time (in milliseconds) that the provider has spent processing once statistics have been enabled, including the time spent waiting for replies from the server as well as the time spent executing code in the provider itself. The classes that include timing code are: SqlConnection SqlCommand SqlDataReader SqlDataAdapter SqlTransaction SqlCommandBuilder To keep performance-critical members as small as possible, the following members are not timed: SqlDataReader this[] operator (all overloads) GetBoolean GetChar GetDateTime GetDecimal GetDouble GetFloat

KEY NAME	DESCRIPTION
	GetGuid GetInt16 GetInt32 GetInt64 GetName GetOrdinal GetSqlBinary GetSqlBoolean GetSqlByte GetSqlDateTime GetSqlDecimal GetSqlDouble GetSqlGuid GetSqlInt16 GetSqlInt32 GetSqlInt64 GetSqlMoney GetSqlSingle GetSqlString GetString IsDBNull
`IduCount`	Returns the total number of INSERT, DELETE, and UPDATE statements executed through the connection once the application has started using the provider and has enabled statistics.
`IduRows`	Returns the total number of rows affected by INSERT, DELETE, and UPDATE statements executed through the connection once the application has started using the provider and has enabled statistics.
`NetworkServerTime`	Returns the cumulative amount of time (in milliseconds) that the provider spent waiting for replies from the server once the application has started using the provider and has enabled statistics.
`PreparedExecs`	Returns the number of prepared commands executed through the connection once the application has started using the provider and has enabled statistics.

KEY NAME	DESCRIPTION
`Prepares`	Returns the number of statements prepared through the connection once the application has started using the provider and has enabled statistics.
`SelectCount`	Returns the number of SELECT statements executed through the connection once the application has started using the provider and has enabled statistics. This includes FETCH statements to retrieve rows from cursors, and the count for SELECT statements is updated when the end of a SqlDataReader is reached.
`SelectRows`	Returns the number of rows selected once the application has started using the provider and has enabled statistics. This counter reflects all the rows generated by SQL statements, even those that were not actually consumed by the caller. For example, closing a data reader before reading the entire result set would not affect the count. This includes the rows retrieved from cursors through FETCH statements.
`ServerRoundtrips`	Returns the number of times the connection sent commands to the server and got a reply back once the application has started using the provider and has enabled statistics.
`SumResultSets`	Returns the number of result sets that have been used once the application has started using the provider and has enabled statistics. For example this would include any result set returned to the client. For cursors, each fetch or block-fetch operation is considered an independent result set.
`Transactions`	Returns the number of user transactions started once the application has started using the provider and has enabled statistics, including rollbacks. If a connection is running with auto commit on, each command is considered a transaction. This counter increments the transaction count as soon as a BEGIN TRAN statement is executed, regardless of whether the transaction is committed or rolled back later.
`UnpreparedExecs`	Returns the number of unprepared statements executed through the connection once the application has started using the provider and has enabled statistics.

Retrieving a Value

The following console application shows how to enable statistics on a connection, retrieve four individual statistic values, and write them out to the console window.

> **NOTE**
>
> The following example uses the sample **AdventureWorks** database included with SQL Server. The connection string provided in the sample code assumes the database is installed and available on the local computer. Modify the connection string as necessary for your environment.

```
Option Strict On

Imports System
Imports System.Collections
```

```
Imports System.Data
Imports System.Data.SqlClient

Module Module1

  Sub Main()
    Dim connectionString As String = GetConnectionString()

    Using awConnection As New SqlConnection(connectionString)
      ' StatisticsEnabled is False by default.
      ' It must be set to True to start the
      ' statistic collection process.
      awConnection.StatisticsEnabled = True

      Dim productSQL As String = "SELECT * FROM Production.Product"
      Dim productAdapter As _
          New SqlDataAdapter(productSQL, awConnection)

      Dim awDataSet As New DataSet()

      awConnection.Open()

      productAdapter.Fill(awDataSet, "ProductTable")

      ' Retrieve the current statistics as
      ' a collection of values at this point
      ' and time.
      Dim currentStatistics As IDictionary = _
          awConnection.RetrieveStatistics()

      Console.WriteLine("Total Counters: " & _
          currentStatistics.Count.ToString())
      Console.WriteLine()

      ' Retrieve a few individual values
      ' related to the previous command.
      Dim bytesReceived As Long = _
          CLng(currentStatistics.Item("BytesReceived"))
      Dim bytesSent As Long = _
          CLng(currentStatistics.Item("BytesSent"))
      Dim selectCount As Long = _
          CLng(currentStatistics.Item("SelectCount"))
      Dim selectRows As Long = _
          CLng(currentStatistics.Item("SelectRows"))

      Console.WriteLine("BytesReceived: " & bytesReceived.ToString())
      Console.WriteLine("BytesSent: " & bytesSent.ToString())
      Console.WriteLine("SelectCount: " & selectCount.ToString())
      Console.WriteLine("SelectRows: " & selectRows.ToString())

      Console.WriteLine()
      Console.WriteLine("Press any key to continue")
      Console.ReadLine()
    End Using

  End Sub

  Function GetConnectionString() As String
    ' To avoid storing the connection string in your code,
    ' you can retrieve it from a configuration file.
    Return "Data Source=localhost;Integrated Security=SSPI;" & _
      "Initial Catalog=AdventureWorks"
  End Function
End Module
```

```
using System;
using System.Collections;
```

```
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;

namespace CS_Stats_Console_GetValue
{
  class Program
  {
    static void Main(string[] args)
    {
      string connectionString = GetConnectionString();

      using (SqlConnection awConnection =
        new SqlConnection(connectionString))
      {
        // StatisticsEnabled is False by default.
        // It must be set to True to start the
        // statistic collection process.
        awConnection.StatisticsEnabled = true;

        string productSQL = "SELECT * FROM Production.Product";
        SqlDataAdapter productAdapter =
          new SqlDataAdapter(productSQL, awConnection);

        DataSet awDataSet = new DataSet();

        awConnection.Open();

        productAdapter.Fill(awDataSet, "ProductTable");
        // Retrieve the current statistics as
        // a collection of values at this point
        // and time.
        IDictionary currentStatistics =
          awConnection.RetrieveStatistics();

        Console.WriteLine("Total Counters: " +
          currentStatistics.Count.ToString());
        Console.WriteLine();

        // Retrieve a few individual values
        // related to the previous command.
        long bytesReceived =
            (long) currentStatistics["BytesReceived"];
        long bytesSent =
            (long) currentStatistics["BytesSent"];
        long selectCount =
            (long) currentStatistics["SelectCount"];
        long selectRows =
            (long) currentStatistics["SelectRows"];

        Console.WriteLine("BytesReceived: " +
            bytesReceived.ToString());
        Console.WriteLine("BytesSent: " +
            bytesSent.ToString());
        Console.WriteLine("SelectCount: " +
            selectCount.ToString());
        Console.WriteLine("SelectRows: " +
            selectRows.ToString());

        Console.WriteLine();
        Console.WriteLine("Press any key to continue");
        Console.ReadLine();
      }

    }
    private static string GetConnectionString()
    {
      // To avoid storing the connection string in your code,
      // you can retrieve it from a configuration file.
```

```
        return "Data Source=localhost;Integrated Security=SSPI;" +
          "Initial Catalog=AdventureWorks";
      }
    }
  }
```

Retrieving All Values

The following console application shows how to enable statistics on a connection, retrieve all available statistic values using the enumerator, and write them to the console window.

> **NOTE**
>
> The following example uses the sample **AdventureWorks** database included with SQL Server. The connection string provided in the sample code assumes the database is installed and available on the local computer. Modify the connection string as necessary for your environment.

```
Option Strict On

Imports System
Imports System.Collections
Imports System.Data
Imports System.Data.SqlClient

Module Module1
  Sub Main()
    Dim connectionString As String = GetConnectionString()

    Using awConnection As New SqlConnection(connectionString)
      ' StatisticsEnabled is False by default.
      ' It must be set to True to start the
      ' statistic collection process.
      awConnection.StatisticsEnabled = True

      Dim productSQL As String = "SELECT * FROM Production.Product"
      Dim productAdapter As _
          New SqlDataAdapter(productSQL, awConnection)

      Dim awDataSet As New DataSet()

      awConnection.Open()

      productAdapter.Fill(awDataSet, "ProductTable")

      ' Retrieve the current statistics as
      ' a collection of values at this point
      ' and time.
      Dim currentStatistics As IDictionary = _
          awConnection.RetrieveStatistics()

      Console.WriteLine("Total Counters: " & _
          currentStatistics.Count.ToString())
      Console.WriteLine()

      Console.WriteLine("Key Name and Value")

      ' Note the entries are unsorted.
      For Each entry As DictionaryEntry In currentStatistics
        Console.WriteLine(entry.Key.ToString() & _
            ": " & entry.Value.ToString())
      Next

      Console.WriteLine()
      Console.WriteLine("Press any key to continue")
      Console.ReadLine()
    End Using

  End Sub

  Function GetConnectionString() As String
    ' To avoid storing the connection string in your code,
    ' you can retrieve it from a configuration file.
    Return "Data Source=localhost;Integrated Security=SSPI;" & _
      "Initial Catalog=AdventureWorks"
  End Function
End Module
```

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;
```

```
namespace CS_Stats_Console_GetAll
{
  class Program
  {
    static void Main(string[] args)
    {
      string connectionString = GetConnectionString();

      using (SqlConnection awConnection =
        new SqlConnection(connectionString))
      {
        // StatisticsEnabled is False by default.
        // It must be set to True to start the
        // statistic collection process.
        awConnection.StatisticsEnabled = true;

        string productSQL = "SELECT * FROM Production.Product";
        SqlDataAdapter productAdapter =
            new SqlDataAdapter(productSQL, awConnection);

        DataSet awDataSet = new DataSet();

        awConnection.Open();

        productAdapter.Fill(awDataSet, "ProductTable");

        // Retrieve the current statistics as
        // a collection of values at this point
        // and time.
        IDictionary currentStatistics =
            awConnection.RetrieveStatistics();

        Console.WriteLine("Total Counters: " +
            currentStatistics.Count.ToString());
        Console.WriteLine();

        Console.WriteLine("Key Name and Value");

        // Note the entries are unsorted.
        foreach (DictionaryEntry entry in currentStatistics)
        {
          Console.WriteLine(entry.Key.ToString() +
              ": " + entry.Value.ToString());
        }

        Console.WriteLine();
        Console.WriteLine("Press any key to continue");
        Console.ReadLine();
      }

    }
    private static string GetConnectionString()
    {
      // To avoid storing the connection string in your code,
      // you can retrieve it from a configuration file.
      return "Data Source=localhost;Integrated Security=SSPI;" +
        "Initial Catalog=AdventureWorks";
    }
  }
}
```

See also

- SQL Server and ADO.NET
- ADO.NET Overview

SQL Server Express User Instances

4/26/2022 • 8 minutes to read • Edit Online

Microsoft SQL Server Express Edition (SQL Server Express) supports the user instance feature, which is only available when using the .NET Framework Data Provider for SQL Server (`SqlClient`). A user instance is a separate instance of the SQL Server Express Database Engine that is generated by a parent instance. User instances allow users who are not administrators on their local computers to attach and connect to SQL Server Express databases. Each instance runs under the security context of the individual user, on a one-instance-per-user basis.

User Instance Capabilities

User instances are useful for users who are running Windows under a least-privilege user account (LUA). Each user has SQL Server system administrator (`sysadmin`) privileges over the instance running on their computer without needing to run as a Windows administrator as well. Software executing on a user instance with limited permissions cannot make system-wide changes because the instance of SQL Server Express is running under the non-administrator Windows account of the user, not as a service. Each user instance is isolated from its parent instance and from any other user instances running on the same computer. Databases running on a user instance are opened in single-user mode only, and it is not possible for multiple users to connect to databases running on a user instance. Replication and distributed queries are also disabled for user instances.

> **NOTE**
> User instances are not needed for users who are already administrators on their own computers, or for scenarios involving multiple database users.

Enabling User Instances

To generate user instances, a parent instance of SQL Server Express must be running. User instances are enabled by default when SQL Server Express is installed, and they can be explicitly enabled or disabled by a system administrator executing the **sp_configure** system stored procedure on the parent instance.

```
-- Enable user instances.
sp_configure 'user instances enabled','1'

-- Disable user instances.
sp_configure 'user instances enabled','0'
```

The network protocol for user instances must be local Named Pipes. A user instance cannot be started on a remote instance of SQL Server, and SQL Server logins are not allowed.

Connecting to a User Instance

The `User Instance` and `AttachDBFilename` ConnectionString keywords allow a SqlConnection to connect to a user instance. User instances are also supported by the SqlConnectionStringBuilder `UserInstance` and `AttachDBFilename` properties.

Note the following about the sample connection string shown below:

- The `Data Source` keyword refers to the parent instance of SQL Server Express that is generating the user

instance. The default instance is .\sqlexpress.

- `Integrated Security` is set to `true`. To connect to a user instance, Windows Authentication is required; SQL Server logins are not supported.
- The `User Instance` is set to `true`, which invokes a user instance. (The default is `false`.)
- The `AttachDbFileName` connection string keyword is used to attach the primary database file (.mdf), which must include the full path name. `AttachDbFileName` also corresponds to the "extended properties" and "initial file name" keys within a SqlConnection connection string.
- The `|DataDirectory|` substitution string enclosed in the pipe symbols refers to the data directory of the application opening the connection and provides a relative path indicating the location of the .mdf and .ldf database and log files. If you want to locate these files elsewhere, you must provide the full path to the files.

```
Data Source=.\\SQLExpress;Integrated Security=true;
User Instance=true;AttachDBFilename=|DataDirectory|\InstanceDB.mdf;
Initial Catalog=InstanceDB;
```

NOTE

You can also use the SqlConnectionStringBuilderUserInstance and AttachDBFilename properties to build a connection string at run time.

Using the |DataDirectory| Substitution String

`AttachDbFileName` was extended in ADO.NET 2.0 with the introduction of the `|DataDirectory|` (enclosed in pipe symbols) substitution string. `DataDirectory` is used in conjunction with `AttachDbFileName` to indicate a relative path to a data file, allowing developers to create connection strings that are based on a relative path to the data source instead of being required to specify a full path.

The physical location that `DataDirectory` points to depends on the type of application. In this example, the Northwind.mdf file to be attached is located in the application's \app_data folder.

```
Data Source=.\\SQLExpress;Integrated Security=true;
User Instance=true;
AttachDBFilename=|DataDirectory|\app_data\Northwind.mdf;
Initial Catalog=Northwind;
```

When `DataDirectory` is used, the resulting file path cannot be higher in the directory structure than the directory pointed to by the substitution string. For example, if the fully expanded `DataDirectory` is C:\AppDirectory\app_data, then the sample connection string shown above works because it is below c:\AppDirectory. However, attempting to specify `DataDirectory` as `|DataDirectory|\..\data` will result in an error because \data is not a subdirectory of \AppDirectory.

If the connection string has an improperly formatted substitution string, an ArgumentException will be thrown.

NOTE

System.Data.SqlClient resolves the substitution strings into full paths against the local computer file system. Therefore, remote server, HTTP, and UNC path names are not supported. An exception is thrown when the connection is opened if the server is not located on the local computer.

When the SqlConnection is opened, it is redirected from the default SQL Server Express instance to a run-time

initiated instance running under the caller's account.

> **NOTE**
> It may be necessary to increase the ConnectionTimeout value since user instances may take longer to load than regular instances.

The following code fragment opens a new `SqlConnection`, displays the connection string in the console window, and then closes the connection when exiting the `using` code block.

```
Private Sub OpenSqlConnection()
    ' Retrieve the connection string.
    Dim connectionString As String = GetConnectionString()

    Using connection As New SqlConnection(connectionString)
        connection.Open()
        Console.WriteLine("ConnectionString: {0}", _
           connection.ConnectionString)
    End Using
End Sub
```

```
private static void OpenSqlConnection()
{
    // Retrieve the connection string.
    string connectionString = GetConnectionString();

    using (SqlConnection connection =
        new SqlConnection(connectionString))
    {
        connection.Open();
        Console.WriteLine("ConnectionString: {0}",
             connection.ConnectionString);
    }
}
```

> **NOTE**
> User instances are not supported in common language runtime (CLR) code that is running inside of SQL Server. An InvalidOperationException is thrown if `Open` is called on a SqlConnection that has `User Instance=true` in the connection string.

Lifetime of a User Instance Connection

Unlike versions of SQL Server that run as a service, SQL Server Express instances do not need to be manually started and stopped. Each time a user logs in and connects to a user instance, the user instance is started if it is not already running. User instance databases have the `AutoClose` option set so that the database is automatically shut down after a period of inactivity. The sqlservr.exe process that is started is kept running for a limited time-out period after the last connection to the instance is closed, so it does not need to be restarted if another connection is opened before the time-out has expired. The user instance automatically shuts down if no new connection opens before that time-out period has expired. A system administrator on the parent instance can set the duration of the time-out period for a user instance by using **sp_configure** to change the **user instance timeout** option. The default is 60 minutes.

> **NOTE**
>
> If `Min Pool Size` is used in the connection string with a value greater than zero, the connection pooler will always maintain a few opened connections, and the user instance will not automatically shut down.

How User Instances Work

The first time a user instance is generated for each user, the **master** and **msdb** system databases are copied from the Template Data folder to a path under the user's local application data repository directory for exclusive use by the user instance. This path is typically `C:\Documents and Settings\<UserName>\Local Settings\Application Data\Microsoft\Microsoft SQL Server Data\SQLEXPRESS`. When a user instance starts up, the **tempdb**, log, and trace files are also written to this directory. A name is generated for the instance, which is guaranteed to be unique for each user.

By default all members of the Windows Builtin\Users group are granted permissions to connect on the local instance as well as read and execute permissions on the SQL Server binaries. Once the credentials of the calling user hosting the user instance have been verified, that user becomes the `sysadmin` on that instance. Only shared memory is enabled for user instances, which means that only operations on the local machine are possible.

Users must be granted both read and write permissions on the .mdf and .ldf files specified in the connection string.

> **NOTE**
>
> The .mdf and .ldf files represent the database and log files, respectively. These two files are a matched set, so care must be taken during backup and restore operations. The database file contains information about the exact version of the log file, and the database will not open if it is coupled with the wrong log file.

To avoid data corruption, a database in the user instance is opened with exclusive access. If two different user instances share the same database on the same computer, the user on the first instance must close the database before it can be opened in a second instance.

User Instance Scenarios

User instances provide developers of database applications with a SQL Server data store that does not depend on developers having administrative accounts on their development computers. User instances are based on the Access/Jet model, where the database application simply connects to a file, and the user automatically has full permissions on all of the database objects without needing the intervention of a system administrator to grant permissions. It is intended to work in situations where the user is running under a least-privilege user account (LUA) and does not have administrative privileges on the server or local machine, yet needs to create database objects and applications. User instances allow users to create instances at run time that run under the user's own security context, and not in the security context of a more privileged system service.

> **IMPORTANT**
>
> User instances should only be used in scenarios where all the applications using it are fully trusted.

User instance scenarios include:

- Any single-user application where sharing data is not required.
- ClickOnce deployment. If .NET Framework 2.0 (or later) and SQL Server Express are already installed on

the target computer, the installation package downloaded as a result of a ClickOnce action can be installed and used by non-administrator users. Note that an administrator must install SQL Server Express if that is part of the setup. For more information, see ClickOnce Deployment for Windows Forms.

- Dedicated ASP.NET hosting using Windows Authentication. A single SQL Server Express instance can be hosted on an intranet. The application connects using the ASPNET Windows account, not by using impersonation. User instances should not be used for third-party or shared hosting scenarios where all applications would share the same user instance and would no longer remain isolated from each other.

See also

- SQL Server and ADO.NET
- Connection Strings
- Connecting to a Data Source
- ADO.NET Overview

Database Mirroring in SQL Server

4/26/2022 • 3 minutes to read • Edit Online

Database mirroring in SQL Server allows you to keep a copy, or mirror, of a SQL Server database on a standby server. Mirroring ensures that two separate copies of the data exist at all times, providing high availability and complete data redundancy. The .NET Data Provider for SQL Server provides implicit support for database mirroring, so that the developer does not need to take any action or write any code once it has been configured for a SQL Server database. In addition, the SqlConnection object supports an explicit connection mode that allows supplying the name of a failover partner server in the ConnectionString.

The following simplified sequence of events occurs for a SqlConnection object that targets a database configured for mirroring:

1. The client application successfully connects to the principal database, and the server sends back the name of the partner server, which is then cached on the client.
2. If the server containing the principal database fails or connectivity is interrupted, connection and transaction state is lost. The client application attempts to re-establish a connection to the principal database and fails.
3. The client application then transparently attempts to establish a connection to the mirror database on the partner server. If it succeeds, the connection is redirected to the mirror database, which then becomes the new principal database.

Specifying the Failover Partner in the Connection String

If you supply the name of a failover partner server in the connection string, the client will transparently attempt a connection with the failover partner if the principal database is unavailable when the client application first connects.

```
";Failover Partner=PartnerServerName"
```

If you omit the name of the failover partner server and the principal database is unavailable when the client application first connects then a SqlException is raised.

When a SqlConnection is successfully opened, the failover partner name is returned by the server and supersedes any values supplied in the connection string.

> **NOTE**
>
> You must explicitly specify the initial catalog or database name in the connection string for database mirroring scenarios. If the client receives failover information on a connection that doesn't have an explicitly specified initial catalog or database, the failover information is not cached and the application does not attempt to fail over if the principal server fails. If a connection string has a value for the failover partner, but no value for the initial catalog or database, an `InvalidArgumentException` is raised.

Retrieving the Current Server Name

In the event of a failover, you can retrieve the name of the server to which the current connection is actually connected by using the DataSource property of a SqlConnection object. The following code fragment retrieves

the name of the active server, assuming that the connection variable references an open SqlConnection.

When a failover event occurs and the connection is switched to the mirror server, the **DataSource** property is updated to reflect the mirror name.

```
Dim activeServer As String = connection.DataSource
```

```
string activeServer = connection.DataSource;
```

SqlClient Mirroring Behavior

The client always tries to connect to the current principal server. If it fails, it tries the failover partner. If the mirror database has already been switched to the principal role on the partner server, the connection succeeds and the new principal-mirror mapping is sent to the client and cached for the lifetime of the calling AppDomain. It is not stored in persistent storage and is not available for subsequent connections in a different **AppDomain** or process. However, it is available for subsequent connections within the same **AppDomain**. Note that another **AppDomain** or process running on the same or a different computer always has its pool of connections, and those connections are not reset. In that case, if the primary database goes down, each process or **AppDomain** fails once, and the pool is automatically cleared.

> **NOTE**
> Mirroring support on the server is configured on a per-database basis. If data manipulation operations are executed against other databases not included in the principal/mirror set, either by using multipart names or by changing the current database, the changes to these other databases do not propagate in the event of failure. No error is generated when data is modified in a database that is not mirrored. The developer must evaluate the possible impact of such operations.

Database Mirroring Resources

For conceptual documentation and information on configuring, deploying and administering mirroring, see the following resources in SQL Server documentation.

RESOURCE	DESCRIPTION
Database Mirroring	Describes how to set up and configure mirroring in SQL Server.

See also

- ADO.NET Overview

SQL Server Common Language Runtime Integration

4/26/2022 • 2 minutes to read • Edit Online

SQL Server 2005 introduced the integration of the common language runtime (CLR) component of the .NET Framework for Microsoft Windows. This means that you can write stored procedures, triggers, user-defined types, user-defined functions, user-defined aggregates, and streaming table-valued functions, using any .NET Framework language, including Microsoft Visual Basic .NET and Microsoft Visual C#. The Microsoft.SqlServer.Server namespace contains a set of new application programming interfaces (APIs) so that managed code can interact with the Microsoft SQL Server environment.

This section describes features and behaviors that are specific to SQL Server common language runtime (CLR) integration and the SQL Server in-process specific extensions to ADO.NET.

This section is meant to provide only enough information to get started programming with SQL Server CLR integration, and is not meant to be comprehensive. For more detailed information, see Common Language Runtime (CLR) Integration Programming Concepts.

In This Section

Introduction to SQL Server CLR Integration
Provides an introduction to SQL Server CLR integration. Provides links to additional topics.

CLR User-Defined Functions
Describes how to implement and use the various types of CLR functions: table-valued, scalar, and user-defined aggregate functions.

CLR User-Defined Types
Describes how to implement and use CLR user-defined types. Provides links to additional topics.

CLR Stored Procedures
Describes how to implement and use CLR stored procedures. Provides links to additional topics.

CLR Triggers
Describes how to implement and use CLR triggers. Provides links to additional topics.

The Context Connection
Describes the context connection.

SQL Server In-Process-Specific Behavior of ADO.NET
Describes the SQL Server in-process specific extensions to ADO.NET, and the context connection. Provides links to additional topics.

See also

- SQL Server and ADO.NET
- ADO.NET Overview

Introduction to SQL Server CLR Integration

4/26/2022 • 2 minutes to read • Edit Online

The common language runtime (CLR) is the heart of the Microsoft .NET Framework and provides the execution environment for all .NET Framework code. Code that runs within the CLR is referred to as managed code. The CLR provides various functions and services required for program execution, including just-in-time (JIT) compilation, allocating and managing memory, enforcing type safety, exception handling, thread management, and security.

With the CLR hosted in Microsoft SQL Server (called CLR integration), you can author stored procedures, triggers, user-defined functions, user-defined types, and user-defined aggregates in managed code. Because managed code compiles to native code prior to execution, you can achieve significant performance increases in some scenarios.

Managed code uses Code Access Security (CAS), code links, and application domains to prevent assemblies from performing certain operations. SQL Server uses CAS to help secure the managed code and prevent compromise of the operating system or database server.

This section is meant to provide only enough information to get started programming with SQL Server CLR integration, and is not meant to be comprehensive. For more detailed information, see Common Language Runtime (CLR) Integration Overview.

Enabling CLR Integration

The common language runtime (CLR) integration feature is off by default in Microsoft SQL Server, and must be enabled in order to use objects that are implemented using CLR integration. To enable CLR integration using Transact-SQL, use the `clr enabled` option of the `sp_configure` stored procedure as shown:

```
sp_configure 'clr enabled', 1
GO
RECONFIGURE
GO
```

You can disable CLR integration by setting the `clr enabled` option to 0. When you disable CLR integration, SQL Server stops executing all CLR routines and unloads all application domains.

For more detailed information, see Enabling CLR Integration.

Deploying a CLR Assembly

Once the CLR methods have been tested and verified on the test server, they can be distributed to production servers using a deployment script. The deployment script can be generated manually, or by using SQL Server Management Studio. For more detailed information, see the version of SQL Server documentation for the version of SQL Server you are using.

SQL Server documentation

1. Deploying CLR Database Objects

CLR Integration Security

The security model of the Microsoft SQL Server integration with the Microsoft .NET Framework common

language runtime (CLR) manages and secures access between different types of CLR and non-CLR objects running within SQL Server. These objects may be called by a Transact-SQL statement or another CLR object running in the server.

For more detailed information, see CLR Integration Security.

Debugging a CLR Assembly

Microsoft SQL Server provides support for debugging Transact-SQL and common language runtime (CLR) objects in the database. Debugging works across languages: users can step seamlessly into CLR objects from Transact-SQL, and vice versa.

For more detailed information, see Debugging CLR Database Objects.

See also

- Code Access Security and ADO.NET
- ADO.NET Overview

CLR User-Defined Functions

4/26/2022 • 2 minutes to read • Edit Online

User-defined functions are routines that can take parameters, perform calculations or other actions, and return a result. You can write user-defined functions in any Microsoft .NET Framework programming language, such as Microsoft Visual Basic .NET or Microsoft Visual C#.

For more detailed information, see CLR User-Defined Functions.

See also

- SQL Server Common Language Runtime Integration
- ADO.NET Overview

CLR User-Defined Types

4/26/2022 • 2 minutes to read • Edit Online

Microsoft SQL Server provides support for user-defined types (UDTs) implemented with the Microsoft .NET Framework common language runtime (CLR). The CLR is integrated into SQL Server, and this mechanism enables you to extend the type system of the database. UDTs provide user extensibility of the SQL Server data type system, and also the ability to define complex structured types.

UDTs can provide two key benefits from an application architecture perspective:

- Strong encapsulation (both in the client and the server) between the internal state and the external behaviors.
- Deep integration with other related server features. Once you define your own UDT, you can use it in all contexts where you can use a system type in SQL Server, including column definitions, and as variables, parameters, function results, cursors, triggers, and replication.

For more detailed information, see the SQL Server documentation for the version of SQL Server you're using.

SQL Server documentation

1. CLR User-Defined Types

See also

- ADO.NET Overview

CLR Stored Procedures

4/26/2022 • 2 minutes to read • Edit Online

Stored procedures are routines that cannot be used in scalar expressions. They can return tabular results and messages to the client, invoke data definition language (DDL) and data manipulation language (DML) statements, and return output parameters.

> **NOTE**
>
> Microsoft Visual Basic does not support output parameters in the same way that Microsoft Visual C# does. You must specify to pass the parameter by reference and apply the <Out()> attribute to represent an output parameter, as in the following:

```
Public Shared Sub ExecuteToClient( <Out()> ByRef number As Integer)
```

For more detailed information, see the version of SQL Server documentation for the version of SQL Server you're using.

SQL Server documentation

1. CLR Stored Procedures

See also

- Creating SQL Server 2005 Objects In Managed Code
- ADO.NET Overview

CLR triggers

4/26/2022 • 2 minutes to read • Edit Online

A trigger is a special type of stored procedure that automatically runs when a language event executes. Because of the Microsoft SQL Server integration with the .NET Framework common language runtime (CLR), you can use any .NET Framework language to create CLR triggers.

For more information, see Create CLR Triggers.

See also

- ADO.NET Overview

The Context Connection

4/26/2022 • 2 minutes to read • Edit Online

The problem of internal data access is a fairly common scenario. That is, you wish to access the same server on which your common language runtime (CLR) stored procedure or function is executing. One option is to create a connection using SqlConnection, specify a connection string that points to the local server, and open the connection. This requires specifying credentials for logging in. The connection is in a different database session than the stored procedure or function, it may have different `SET` options, it is in a separate transaction, it does not see your temporary tables, and so on. If your managed stored procedure or function code is executing in the SQL Server process, it is because someone connected to that server and executed a SQL statement to invoke it. You probably want the stored procedure or function to execute in the context of that connection, along with its transaction, `SET` options, and so on. This is called the context connection.

The context connection lets you execute Transact-SQL statements in the same context that your code was invoked in the first place. For more detailed information, see The Context Connection.

See also

- ADO.NET Overview

SQL Server In-Process-Specific Behavior of ADO.NET

4/26/2022 • 2 minutes to read • Edit Online

There are four main functional extensions to ADO.NET, found in the Microsoft.SqlServer.Server namespace, that are specifically for in-process use: SqlContext, SqlPipe, SqlTriggerContext, and SqlDataRecord.

For more detailed information, see SQL Server In-Process Specific Extensions to ADO.NET.

See also

- ADO.NET Overview

Query Notifications in SQL Server

4/26/2022 • 2 minutes to read • Edit Online

Built upon the Service Broker infrastructure, query notifications allow applications to be notified when data has changed. This feature is particularly useful for applications that provide a cache of information from a database, such as a Web application, and need to be notified when the source data is changed.

There are three ways you can implement query notifications using ADO.NET:

1. The low-level implementation is provided by the `SqlNotificationRequest` class that exposes server-side functionality, enabling you to execute a command with a notification request.
2. The high-level implementation is provided by the `SqlDependency` class, which is a class that provides a high-level abstraction of notification functionality between the source application and SQL Server, enabling you to use a dependency to detect changes in the server. In most cases, this is the simplest and most effective way to leverage SQL Server notifications capability by managed client applications using the .NET Framework Data Provider for SQL Server.
3. In addition, Web applications built using ASP.NET 2.0 or later can use the `SqlCacheDependency` helper classes.

Query notifications are used for applications that need to refresh displays or caches in response to changes in underlying data. Microsoft SQL Server allows .NET Framework applications to send a command to SQL Server and request notification if executing the same command would produce result sets different from those initially retrieved. Notifications generated at the server are sent through queues to be processed later.

You can set up notifications for SELECT and EXECUTE statements. When using an EXECUTE statement, SQL Server registers a notification for the command executed rather than the EXECUTE statement itself. The command must meet the requirements and limitations for a SELECT statement. When a command that registers a notification contains more than one statement, the Database Engine creates a notification for each statement in the batch.

If you are developing an application where you need reliable sub-second notifications when data changes, review the sections **Planning an Efficient Query Notifications Strategy** and **Alternatives to Query Notifications** in the Planning for Notifications article. For more information about Query Notifications and SQL Server Service Broker, see the following links to articles in the SQL Server documentation.

SQL Server documentation

- Using Query Notifications
- Creating a Query for Notification
- Development (Service Broker)
- Service Broker Developer InfoCenter
- Developer's Guide (Service Broker)

In This Section

Enabling Query Notifications
Discusses how to use query notifications, including the requirements for enabling and using them.

SqlDependency in an ASP.NET Application

Demonstrates how to use query notifications from an ASP.NET application.

Detecting Changes with SqlDependency
Demonstrates how to detect when query results will be different from those originally received.

SqlCommand Execution with a SqlNotificationRequest
Demonstrates configuring a SqlCommand object to work with a query notification.

Reference

SqlNotificationRequest
Describes the SqlNotificationRequest class and all of its members.

SqlDependency
Describes the SqlDependency class and all of its members.

SqlCacheDependency
Describes the SqlCacheDependency class and all of its members.

See also

- SQL Server and ADO.NET
- ADO.NET Overview

Enabling Query Notifications

4/26/2022 • 3 minutes to read • Edit Online

Applications that consume query notifications have a common set of requirements. Your data source must be correctly configured to support SQL query notifications, and the user must have the correct client-side and server-side permissions.

To use query notifications you must:

- Enable query notifications for your database.
- Ensure that the user ID used to connect to the database has the necessary permissions.
- Use a SqlCommand object to execute a valid SELECT statement with an associated notification object—either SqlDependency or SqlNotificationRequest.
- Provide code to process the notification if the data being monitored changes.

Query Notifications Requirements

Query notifications are supported only for SELECT statements that meet a list of specific requirements. The following table provides links to the Service Broker and Query Notifications documentation in the SQL Server docs.

SQL Server documentation

- Creating a Query for Notification
- Security Considerations for Service Broker
- Security and Protection (Service Broker)
- Security Considerations for Notifications Services
- Query Notification Permissions
- International Considerations for Service Broker
- Solution Design Considerations (Service Broker)
- Service Broker Developer InfoCenter
- Developer's Guide (Service Broker)

Enabling Query Notifications to Run Sample Code

To enable Service Broker on the **AdventureWorks** database by using SQL Server Management Studio, execute the following Transact-SQL statement:

```
ALTER DATABASE AdventureWorks SET ENABLE_BROKER;
```

For the query notification samples to run correctly, the following Transact-SQL statements must be executed on the database server.

```
CREATE QUEUE ContactChangeMessages;

CREATE SERVICE ContactChangeNotifications
  ON QUEUE ContactChangeMessages
([http://schemas.microsoft.com/SQL/Notifications/PostQueryNotification]);
```

Query Notifications Permissions

Users who execute commands requesting notification must have SUBSCRIBE QUERY NOTIFICATIONS database permission on the server.

Client-side code that runs in a partial trust situation requires the SqlClientPermission.

The following code creates a SqlClientPermission object, setting the PermissionState to Unrestricted. The Demand will force a SecurityException at run time if all callers higher in the call stack have not been granted the permission.

```
// Code requires directives to
// System.Security.Permissions and
// System.Data.SqlClient

private bool CanRequestNotifications()
{
    SqlClientPermission permission =
        new SqlClientPermission(
        PermissionState.Unrestricted);
    try
    {
        permission.Demand();
        return true;
    }
    catch (System.Exception)
    {
        return false;
    }
}
```

```
' Code requires directives to
' System.Security.Permissions and
' System.Data.SqlClient

Private Function CanRequestNotifications() As Boolean

    Dim permission As New SqlClientPermission( _
      PermissionState.Unrestricted)

    Try
        permission.Demand()
        Return True
    Catch ex As Exception
        Return False
    End Try

End Function
```

Choosing a Notification Object

The query notifications API provides two objects to process notifications: SqlDependency and SqlNotificationRequest. In general, most non-ASP.NET applications should use the SqlDependency object.

ASP.NET applications should use the higher-level SqlCacheDependency, which wraps SqlDependency and provides a framework for administering the notification and cache objects.

Using SqlDependency

To use SqlDependency, Service Broker must be enabled for the SQL Server database being used, and users must have permissions to receive notifications. Service Broker objects, such as the notification queue, are predefined.

In addition, SqlDependency automatically launches a worker thread to process notifications as they are posted to the queue; it also parses the Service Broker message, exposing the information as event argument data. SqlDependency must be initialized by calling the `Start` method to establish a dependency to the database. This is a static method that needs to be called only once during application initialization for each database connection required. The `Stop` method should be called at application termination for each dependency connection that was made.

Using SqlNotificationRequest

In contrast, SqlNotificationRequest requires you to implement the entire listening infrastructure yourself. In addition, all the supporting Service Broker objects such as the queue, service, and message types supported by the queue must be defined. This manual approach is useful if your application requires special notification messages or notification behaviors, or if your application is part of a larger Service Broker application.

See also

- Query Notifications in SQL Server
- ADO.NET Overview

SqlDependency in an ASP.NET Application

4/26/2022 • 4 minutes to read • Edit Online

The example in this section shows how to use SqlDependency indirectly by leveraging the ASP.NET SqlCacheDependency object. The SqlCacheDependency object uses a SqlDependency to listen for notifications and correctly update the cache.

> **NOTE**
>
> The sample code assumes that you have enabled query notifications by executing the scripts in Enabling Query Notifications.

About the Sample Application

The sample application uses a single ASP.NET Web page to display product information from the **AdventureWorks** SQL Server database in a GridView control. When the page loads, the code writes the current time to a Label control. It then defines a SqlCacheDependency object and sets properties on the Cache object to store the cache data for up to three minutes. The code then connects to the database and retrieves the data. When the page is loaded and the application is running ASP.NET will retrieve data from the cache, which you can verify by noting that the time on the page does not change. If the data being monitored changes, ASP.NET invalidates the cache and repopulate the `GridView` control with fresh data, updating the time displayed in the `Label` control.

Creating the Sample Application

Follow these steps to create and run the sample application:

1. Create a new ASP.NET Web site.
2. Add a Label and a GridView control to the Default.aspx page.
3. Open the page's class module and add the following directives:

```
Option Strict On
Option Explicit On

Imports System.Data.SqlClient
```

```
using System.Data.SqlClient;
using System.Web.Caching;
```

4. Add the following code in the page's `Page_Load` event:

```
protected void Page_Load(object sender, EventArgs e)
{
    Label1.Text = "Cache Refresh: " +
    DateTime.Now.ToLongTimeString();

    // Create a dependency connection to the database.
    SqlDependency.Start(GetConnectionString());

    using (SqlConnection connection =
        new SqlConnection(GetConnectionString()))
    {
        using (SqlCommand command =
            new SqlCommand(GetSQL(), connection))
        {
            SqlCacheDependency dependency =
                new SqlCacheDependency(command);
            // Refresh the cache after the number of minutes
            // listed below if a change does not occur.
            // This value could be stored in a configuration file.
            int numberOfMinutes = 3;
            DateTime expires =
                DateTime.Now.AddMinutes(numberOfMinutes);

            Response.Cache.SetExpires(expires);
            Response.Cache.SetCacheability(HttpCacheability.Public);
            Response.Cache.SetValidUntilExpires(true);

            Response.AddCacheDependency(dependency);

            connection.Open();

            GridView1.DataSource = command.ExecuteReader();
            GridView1.DataBind();
        }
    }
}
```

```
Protected Sub Page_Load(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles Me.Load

    Label1.Text = "Cache Refresh: " & _
       Date.Now.ToLongTimeString()

    ' Create a dependency connection to the database
    SqlDependency.Start(GetConnectionString())

    Using connection As New SqlConnection(GetConnectionString())
        Using command As New SqlCommand(GetSQL(), connection)
            Dim dependency As New SqlCacheDependency(command)

            ' Refresh the cache after the number of minutes
            ' listed below if a change does not occur.
            ' This value could be stored in a configuration file.
            Dim numberOfMinutes As Integer = 3
            Dim expires As Date = _
                DateTime.Now.AddMinutes(numberOfMinutes)

            Response.Cache.SetExpires(expires)
            Response.Cache.SetCacheability(HttpCacheability.Public)
            Response.Cache.SetValidUntilExpires(True)

            Response.AddCacheDependency(dependency)

            connection.Open()

            GridView1.DataSource = command.ExecuteReader()
            GridView1.DataBind()
        End Using
    End Using
End Sub
```

5. Add two helper methods, `GetConnectionString` and `GetSQL`. The connection string defined uses integrated security. You will need to verify that the account you are using has the necessary database permissions and that the sample database, **AdventureWorks**, has notifications enabled.

```
private string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
    return "Data Source=(local);Integrated Security=true;" +
      "Initial Catalog=AdventureWorks;";
}
private string GetSQL()
{
    return "SELECT Production.Product.ProductID, " +
    "Production.Product.Name, " +
    "Production.Location.Name AS Location, " +
    "Production.ProductInventory.Quantity " +
    "FROM Production.Product INNER JOIN " +
    "Production.ProductInventory " +
    "ON Production.Product.ProductID = " +
    "Production.ProductInventory.ProductID " +
    "INNER JOIN Production.Location " +
    "ON Production.ProductInventory.LocationID = " +
    "Production.Location.LocationID " +
    "WHERE ( Production.ProductInventory.Quantity <= 100 ) " +
    "ORDER BY Production.ProductInventory.Quantity, " +
    "Production.Product.Name;";
}
```

```
Private Function GetConnectionString() As String
    ' To avoid storing the connection string in your code,
    ' you can retrive it from a configuration file.

    Return "Data Source=(local);Integrated Security=true;" & _
     "Initial Catalog=AdventureWorks;"
End Function

Private Function GetSQL() As String
    Return "SELECT Production.Product.ProductID, " & _
    "Production.Product.Name, " & _
    "Production.Location.Name AS Location, " & _
    "Production.ProductInventory.Quantity " & _
    "FROM Production.Product INNER JOIN " & _
    "Production.ProductInventory " & _
    "ON Production.Product.ProductID = " & _
    "Production.ProductInventory.ProductID " & _
    "INNER JOIN Production.Location " & _
    "ON Production.ProductInventory.LocationID = " & _
    "Production.Location.LocationID " & _
    "WHERE ( Production.ProductInventory.Quantity <= 100) " & _
    "ORDER BY Production.ProductInventory.Quantity, " & _
    "Production.Product.Name;"
End Function
```

Testing the Application

The application caches the data displayed on the Web form and refreshes it every three minutes if there is no activity. If a change occurs to the database, the cache is refreshed immediately. Run the application from Visual Studio, which loads the page into the browser. The cache refresh time displayed indicates when the cache was last refreshed. Wait three minutes, and then refresh the page, causing a postback event to occur. Note that the time displayed on the page has changed. If you refresh the page in less than three minutes, the time displayed on the page will remain the same.

Now update the data in the database, using a Transact-SQL UPDATE command and refresh the page. The time displayed now indicates that the cache was refreshed with the new data from the database. Note that although the cache is updated, the time displayed on the page does not change until a postback event occurs.

Distributed cache synchronization using SQL Dependency

Some of the third-party distributed caches such as NCache provide support to synchronize the SQL database and cache using SQL Dependency. For more information and an example source code implementation, see Distributed cache SQL Dependency sample.

See also

- Query Notifications in SQL Server
- ADO.NET Overview

Detecting Changes with SqlDependency

4/26/2022 • 3 minutes to read • Edit Online

A SqlDependency object can be associated with a SqlCommand in order to detect when query results differ from those originally retrieved. You can also assign a delegate to the `OnChange` event, which will fire when the results change for an associated command. You must associate the SqlDependency with the command before you execute the command. The `HasChanges` property of the SqlDependency can also be used to determine if the query results have changed since the data was first retrieved.

Security Considerations

The dependency infrastructure relies on a SqlConnection that is opened when Start is called in order to receive notifications that the underlying data has changed for a given command. The ability for a client to initiate the call to `SqlDependency.Start` is controlled through the use of SqlClientPermission and code access security attributes. For more information, see Enabling Query Notifications and Code Access Security and ADO.NET.

Example

The following steps illustrate how to declare a dependency, execute a command, and receive a notification when the result set changes:

1. Initiate a `SqlDependency` connection to the server.
2. Create SqlConnection and SqlCommand objects to connect to the server and define a Transact-SQL statement.
3. Create a new `SqlDependency` object, or use an existing one, and bind it to the `SqlCommand` object. Internally, this creates a SqlNotificationRequest object and binds it to the command object as needed. This notification request contains an internal identifier that uniquely identifies this `SqlDependency` object. It also starts the client listener if it is not already active.
4. Subscribe an event handler to the `OnChange` event of the `SqlDependency` object.
5. Execute the command using any of the `Execute` methods of the `SqlCommand` object. Because the command is bound to the notification object, the server recognizes that it must generate a notification, and the queue information will point to the dependencies queue.
6. Stop the `SqlDependency` connection to the server.

If any user subsequently changes the underlying data, Microsoft SQL Server detects that there is a notification pending for such a change, and posts a notification that is processed and forwarded to the client through the underlying `SqlConnection` that was created by calling `SqlDependency.Start`. The client listener receives the invalidation message. The client listener then locates the associated `SqlDependency` object and fires the `OnChange` event.

The following code fragment shows the design pattern you would use to create a sample application.

```
Sub Initialization()
    ' Create a dependency connection.
    SqlDependency.Start(connectionString, queueName)
End Sub

Sub SomeMethod()
    ' Assume connection is an open SqlConnection.
    ' Create a new SqlCommand object.
    Using command As New SqlCommand( _
      "SELECT ShipperID, CompanyName, Phone FROM dbo.Shippers", _
      connection)

        ' Create a dependency and associate it with the SqlCommand.
        Dim dependency As New SqlDependency(command)
        ' Maintain the refernce in a class member.
        ' Subscribe to the SqlDependency event.
        AddHandler dependency.OnChange, AddressOf OnDependencyChange

        ' Execute the command.
        Using reader = command.ExecuteReader()
            ' Process the DataReader.
        End Using
    End Using
End Sub

' Handler method
Sub OnDependencyChange(ByVal sender As Object, _
    ByVal e As SqlNotificationEventArgs)
    ' Handle the event (for example, invalidate this cache entry).
End Sub

Sub Termination()
    ' Release the dependency
    SqlDependency.Stop(connectionString, queueName)
End Sub
```

```
void Initialization()
{
    // Create a dependency connection.
    SqlDependency.Start(connectionString, queueName);
}

void SomeMethod()
{
    // Assume connection is an open SqlConnection.

    // Create a new SqlCommand object.
    using (SqlCommand command=new SqlCommand(
        "SELECT ShipperID, CompanyName, Phone FROM dbo.Shippers",
        connection))
    {

        // Create a dependency and associate it with the SqlCommand.
        SqlDependency dependency=new SqlDependency(command);
        // Maintain the reference in a class member.

        // Subscribe to the SqlDependency event.
        dependency.OnChange+=new
           OnChangeEventHandler(OnDependencyChange);

        // Execute the command.
        using (SqlDataReader reader = command.ExecuteReader())
        {
            // Process the DataReader.
        }
    }
}

// Handler method
void OnDependencyChange(object sender,
   SqlNotificationEventArgs e )
{
  // Handle the event (for example, invalidate this cache entry).
}

void Termination()
{
    // Release the dependency.
    SqlDependency.Stop(connectionString, queueName);
}
```

See also

- Query Notifications in SQL Server
- ADO.NET Overview

SqlCommand Execution with a SqlNotificationRequest

4/26/2022 • 2 minutes to read • Edit Online

A SqlCommand can be configured to generate a notification when data changes after it has been fetched from the server and the result set would be different if the query were executed again. This is useful for scenarios where you want to use custom notification queues on the server or when you do not want to maintain live objects.

Creating the Notification Request

You can use a SqlNotificationRequest object to create the notification request by binding it to a `SqlCommand` object. Once the request is created, you no longer need the `SqlNotificationRequest` object. You can query the queue for any notifications and respond appropriately. Notifications can occur even if the application is shut down and subsequently restarted.

When the command with the associated notification is executed, any changes to the original result set trigger sending a message to the SQL Server queue that was configured in the notification request.

How you poll the SQL Server queue and interpret the message is specific to your application. The application is responsible for polling the queue and reacting based on the contents of the message.

> **NOTE**
> When using SQL Server notification requests with SqlDependency, create your own queue name instead of using the default service name.

There are no new client-side security elements for SqlNotificationRequest. This is primarily a server feature, and the server has created special privileges that users must have to request a notification.

Example

The following code fragment demonstrates how to create a SqlNotificationRequest and associate it with a SqlCommand.

```
' Assume connection is an open SqlConnection.
' Create a new SqlCommand object.
Dim command As New SqlCommand( _
  "SELECT ShipperID, CompanyName, Phone FROM dbo.Shippers", connection)

' Create a SqlNotificationRequest object.
Dim notifcationRequest As New SqlNotificationRequest()
notificationRequest.id = "NotificationID"
notificationRequest.Service = "mySSBQueue"

' Associate the notification request with the command.
command.Notification = notificationRequest
' Execute the command.
command.ExecuteReader()
' Process the DataReader.
' You can use Transact-SQL syntax to periodically poll the
' SQL Server queue to see if you have a new message.
```

```
// Assume connection is an open SqlConnection.
// Create a new SqlCommand object.
SqlCommand command=new SqlCommand(
 "SELECT ShipperID, CompanyName, Phone FROM dbo.Shippers", connection);

// Create a SqlNotificationRequest object.
SqlNotificationRequest notificationRequest=new SqlNotificationRequest();
notificationRequest.id="NotificationID";
notificationRequest.Service="mySSBQueue";

// Associate the notification request with the command.
command.Notification=notificationRequest;
// Execute the command.
command.ExecuteReader();
// Process the DataReader.
// You can use Transact-SQL syntax to periodically poll the
// SQL Server queue to see if you have a new message.
```

See also

- Query Notifications in SQL Server
- ADO.NET Overview

Snapshot Isolation in SQL Server

4/26/2022 • 20 minutes to read • Edit Online

Snapshot isolation enhances concurrency for OLTP applications.

Understanding Snapshot Isolation and Row Versioning

Once snapshot isolation is enabled, updated row versions for each transaction must be maintained. Prior to SQL Server 2019, these versions were stored in **tempdb**. SQL Server 2019 introduces a new feature, Accelerated Database Recovery (ADR) which requires its own set of row versions. So, as of SQL Server 2019, if ADR is not enabled, row versions are kept in **tempdb** as always. If ADR is enabled, then all row versions, both related to snapshot isolation and ADR, are kept in ADR's Persistent Version Store (PVS), which is located in the user database in a filegroup which the user specifies. A unique transaction sequence number identifies each transaction, and these unique numbers are recorded for each row version. The transaction works with the most recent row versions having a sequence number before the sequence number of the transaction. Newer row versions created after the transaction has begun are ignored by the transaction.

The term "snapshot" reflects the fact that all queries in the transaction see the same version, or snapshot, of the database, based on the state of the database at the moment in time when the transaction begins. No locks are acquired on the underlying data rows or data pages in a snapshot transaction, which permits other transactions to execute without being blocked by a prior uncompleted transaction. Transactions that modify data do not block transactions that read data, and transactions that read data do not block transactions that write data, as they normally would under the default READ COMMITTED isolation level in SQL Server. This non-blocking behavior also significantly reduces the likelihood of deadlocks for complex transactions.

Snapshot isolation uses an optimistic concurrency model. If a snapshot transaction attempts to commit modifications to data that has changed since the transaction began, the transaction will roll back and an error will be raised. You can avoid this by using UPDLOCK hints for SELECT statements that access data to be modified. For more information, see Hints (Transact-SQL).

Snapshot isolation must be enabled by setting the ALLOW_SNAPSHOT_ISOLATION ON database option before it is used in transactions. This activates the mechanism for storing row versions in the temporary database (**tempdb**). You must enable snapshot isolation in each database that uses it with the Transact-SQL ALTER DATABASE statement. In this respect, snapshot isolation differs from the traditional isolation levels of READ COMMITTED, REPEATABLE READ, SERIALIZABLE, and READ UNCOMMITTED, which require no configuration. The following statements activate snapshot isolation and replace the default READ COMMITTED behavior with SNAPSHOT:

```
ALTER DATABASE MyDatabase
SET ALLOW_SNAPSHOT_ISOLATION ON

ALTER DATABASE MyDatabase
SET READ_COMMITTED_SNAPSHOT ON
```

Setting the READ_COMMITTED_SNAPSHOT ON option allows access to versioned rows under the default READ COMMITTED isolation level. If the READ_COMMITTED_SNAPSHOT option is set to OFF, you must explicitly set the Snapshot isolation level for each session in order to access versioned rows.

Managing Concurrency with Isolation Levels

The isolation level under which a Transact-SQL statement executes determines its locking and row versioning

behavior. An isolation level has connection-wide scope, and once set for a connection with the SET TRANSACTION ISOLATION LEVEL statement, it remains in effect until the connection is closed or another isolation level is set. When a connection is closed and returned to the pool, the isolation level from the last SET TRANSACTION ISOLATION LEVEL statement is retained. Subsequent connections reusing a pooled connection use the isolation level that was in effect at the time the connection is pooled.

Individual queries issued within a connection can contain lock hints that modify the isolation for a single statement or transaction but do not affect the isolation level of the connection. Isolation levels or lock hints set in stored procedures or functions do not change the isolation level of the connection that calls them and are in effect only for the duration of the stored procedure or function call.

Four isolation levels defined in the SQL-92 standard were supported in early versions of SQL Server:

- READ UNCOMMITTED is the least restrictive isolation level because it ignores locks placed by other transactions. Transactions executing under READ UNCOMMITTED can read modified data values that have not yet been committed by other transactions; these are called "dirty" reads.
- READ COMMITTED is the default isolation level for SQL Server. It prevents dirty reads by specifying that statements cannot read data values that have been modified but not yet committed by other transactions. Other transactions can still modify, insert, or delete data between executions of individual statements within the current transaction, resulting in non-repeatable reads, or "phantom" data.
- REPEATABLE READ is a more restrictive isolation level than READ COMMITTED. It encompasses READ COMMITTED and additionally specifies that no other transactions can modify or delete data that has been read by the current transaction until the current transaction commits. Concurrency is lower than for READ COMMITTED because shared locks on read data are held for the duration of the transaction instead of being released at the end of each statement.
- SERIALIZABLE is the most restrictive isolation level, because it locks entire ranges of keys and holds the locks until the transaction is complete. It encompasses REPEATABLE READ and adds the restriction that other transactions cannot insert new rows into ranges that have been read by the transaction until the transaction is complete.

For more information, refer to the Transaction Locking and Row Versioning Guide.

Snapshot Isolation Level Extensions

SQL Server introduced extensions to the SQL-92 isolation levels with the introduction of the SNAPSHOT isolation level and an additional implementation of READ COMMITTED. The READ_COMMITTED_SNAPSHOT isolation level can transparently replace READ COMMITTED for all transactions.

- SNAPSHOT isolation specifies that data read within a transaction will never reflect changes made by other simultaneous transactions. The transaction uses the data row versions that exist when the transaction begins. No locks are placed on the data when it is read, so SNAPSHOT transactions do not block other transactions from writing data. Transactions that write data do not block snapshot transactions from reading data. You need to enable snapshot isolation by setting the ALLOW_SNAPSHOT_ISOLATION database option in order to use it.
- The READ_COMMITTED_SNAPSHOT database option determines the behavior of the default READ COMMITTED isolation level when snapshot isolation is enabled in a database. If you do not explicitly specify READ_COMMITTED_SNAPSHOT ON, READ COMMITTED is applied to all implicit transactions. This produces the same behavior as setting READ_COMMITTED_SNAPSHOT OFF (the default). When READ_COMMITTED_SNAPSHOT OFF is in effect, the Database Engine uses shared locks to enforce the default isolation level. If you set the READ_COMMITTED_SNAPSHOT database option to ON, the database engine uses row versioning and snapshot isolation as the default, instead of using locks to protect the data.

How Snapshot Isolation and Row Versioning Work

When the SNAPSHOT isolation level is enabled, each time a row is updated, the SQL Server Database Engine stores a copy of the original row in **tempdb**, and adds a transaction sequence number to the row. The following is the sequence of events that occurs:

- A new transaction is initiated, and it is assigned a transaction sequence number.
- The Database Engine reads a row within the transaction and retrieves the row version from **tempdb** whose sequence number is closest to, and lower than, the transaction sequence number.
- The Database Engine checks to see if the transaction sequence number is not in the list of transaction sequence numbers of the uncommitted transactions active when the snapshot transaction started.
- The transaction reads the version of the row from **tempdb** that was current as of the start of the transaction. It will not see new rows inserted after the transaction was started because those sequence number values will be higher than the value of the transaction sequence number.
- The current transaction will see rows that were deleted after the transaction began, because there will be a row version in **tempdb** with a lower sequence number value.

The net effect of snapshot isolation is that the transaction sees all of the data as it existed at the start of the transaction, without honoring or placing any locks on the underlying tables. This can result in performance improvements in situations where there is contention.

A snapshot transaction always uses optimistic concurrency control, withholding any locks that would prevent other transactions from updating rows. If a snapshot transaction attempts to commit an update to a row that was changed after the transaction began, the transaction is rolled back, and an error is raised.

Working with Snapshot Isolation in ADO.NET

Snapshot isolation is supported in ADO.NET by the SqlTransaction class. If a database has been enabled for snapshot isolation but is not configured for READ_COMMITTED_SNAPSHOT ON, you must initiate a SqlTransaction using the **IsolationLevel.Snapshot** enumeration value when calling the BeginTransaction method. This code fragment assumes that connection is an open SqlConnection object.

```
Dim sqlTran As SqlTransaction = _
  connection.BeginTransaction(IsolationLevel.Snapshot)
```

```
SqlTransaction sqlTran =
  connection.BeginTransaction(IsolationLevel.Snapshot);
```

Example

The following example demonstrates how the different isolation levels behave by attempting to access locked data, and it is not intended to be used in production code.

The code connects to the **AdventureWorks** sample database in SQL Server and creates a table named **TestSnapshot** and inserts one row of data. The code uses the ALTER DATABASE Transact-SQL statement to turn on snapshot isolation for the database, but it does not set the READ_COMMITTED_SNAPSHOT option, leaving the default READ COMMITTED isolation-level behavior in effect. The code then performs the following actions:

- It begins, but does not complete, sqlTransaction1, which uses the SERIALIZABLE isolation level to start an update transaction. This has the effect of locking the table.
- It opens a second connection and initiates a second transaction using the SNAPSHOT isolation level to read the data in the **TestSnapshot** table. Because snapshot isolation is enabled, this transaction can read

the data that existed before sqlTransaction1 started.

- It opens a third connection and initiates a transaction using the READ COMMITTED isolation level to attempt to read the data in the table. In this case, the code cannot read the data because it cannot read past the locks placed on the table in the first transaction and times out. The same result would occur if the REPEATABLE READ and SERIALIZABLE isolation levels were used because these isolation levels also cannot read past the locks placed in the first transaction.
- It opens a fourth connection and initiates a transaction using the READ UNCOMMITTED isolation level, which performs a dirty read of the uncommitted value in sqlTransaction1. This value may never actually exist in the database if the first transaction is not committed.
- It rolls back the first transaction and cleans up by deleting the **TestSnapshot** table and turning off snapshot isolation for the **AdventureWorks** database.

> **NOTE**
> The following examples use the same connection string with connection pooling turned off. If a connection is pooled, resetting its isolation level does not reset the isolation level at the server. As a result, subsequent connections that use the same pooled inner connection start with their isolation levels set to that of the pooled connection. An alternative to turning off connection pooling is to set the isolation level explicitly for each connection.

```
// Assumes GetConnectionString returns a valid connection string
// where pooling is turned off by setting Pooling=False;.
string connectionString = GetConnectionString();
using (SqlConnection connection1 = new SqlConnection(connectionString))
{
    // Drop the TestSnapshot table if it exists
    connection1.Open();
    SqlCommand command1 = connection1.CreateCommand();
    command1.CommandText = "IF EXISTS "
        + "(SELECT * FROM sys.tables WHERE name=N'TestSnapshot') "
        + "DROP TABLE TestSnapshot";
    try
    {
        command1.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    // Enable Snapshot isolation
    command1.CommandText =
        "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION ON";
    command1.ExecuteNonQuery();

    // Create a table named TestSnapshot and insert one row of data
    command1.CommandText =
        "CREATE TABLE TestSnapshot (ID int primary key, valueCol int)";
    command1.ExecuteNonQuery();
    command1.CommandText =
        "INSERT INTO TestSnapshot VALUES (1,1)";
    command1.ExecuteNonQuery();

    // Begin, but do not complete, a transaction to update the data
    // with the Serializable isolation level, which locks the table
    // pending the commit or rollback of the update. The original
    // value in valueCol was 1, the proposed new value is 22.
    SqlTransaction transaction1 =
        connection1.BeginTransaction(IsolationLevel.Serializable);
    command1.Transaction = transaction1;
    command1.CommandText =
        "UPDATE TestSnapshot SET valueCol=22 WHERE ID=1";
```

```
        "UPDATE TestSnapshot SET valueCol=22 WHERE ID=1";
    command1.ExecuteNonQuery();

    // Open a second connection to AdventureWorks
    using (SqlConnection connection2 = new SqlConnection(connectionString))
    {
        connection2.Open();
        // Initiate a second transaction to read from TestSnapshot
        // using Snapshot isolation. This will read the original
        // value of 1 since transaction1 has not yet committed.
        SqlCommand command2 = connection2.CreateCommand();
        SqlTransaction transaction2 =
            connection2.BeginTransaction(IsolationLevel.Snapshot);
        command2.Transaction = transaction2;
        command2.CommandText =
            "SELECT ID, valueCol FROM TestSnapshot";
        SqlDataReader reader2 = command2.ExecuteReader();
        while (reader2.Read())
        {
            Console.WriteLine("Expected 1,1 Actual "
                + reader2.GetValue(0).ToString()
                + "," + reader2.GetValue(1).ToString());
        }
        transaction2.Commit();
    }

    // Open a third connection to AdventureWorks and
    // initiate a third transaction to read from TestSnapshot
    // using ReadCommitted isolation level. This transaction
    // will not be able to view the data because of
    // the locks placed on the table in transaction1
    // and will time out after 4 seconds.
    // You would see the same behavior with the
    // RepeatableRead or Serializable isolation levels.
    using (SqlConnection connection3 = new SqlConnection(connectionString))
    {
        connection3.Open();
        SqlCommand command3 = connection3.CreateCommand();
        SqlTransaction transaction3 =
            connection3.BeginTransaction(IsolationLevel.ReadCommitted);
        command3.Transaction = transaction3;
        command3.CommandText =
            "SELECT ID, valueCol FROM TestSnapshot";
        command3.CommandTimeout = 4;
        try
        {
            SqlDataReader sqldatareader3 = command3.ExecuteReader();
            while (sqldatareader3.Read())
            {
                Console.WriteLine("You should never hit this.");
            }
            transaction3.Commit();
        }
        catch (Exception ex)
        {
            Console.WriteLine("Expected timeout expired exception: "
                + ex.Message);
            transaction3.Rollback();
        }
    }

    // Open a fourth connection to AdventureWorks and
    // initiate a fourth transaction to read from TestSnapshot
    // using the ReadUncommitted isolation level. ReadUncommitted
    // will not hit the table lock, and will allow a dirty read
    // of the proposed new value 22 for valueCol. If the first
    // transaction rolls back, this value will never actually have
    // existed in the database.
    using (SqlConnection connection4 = new SqlConnection(connectionString))
    {
```

```
    {
        connection4.Open();
        SqlCommand command4 = connection4.CreateCommand();
        SqlTransaction transaction4 =
            connection4.BeginTransaction(IsolationLevel.ReadUncommitted);
        command4.Transaction = transaction4;
        command4.CommandText =
            "SELECT ID, valueCol FROM TestSnapshot";
        SqlDataReader reader4 = command4.ExecuteReader();
        while (reader4.Read())
        {
            Console.WriteLine("Expected 1,22 Actual "
                + reader4.GetValue(0).ToString()
                + "," + reader4.GetValue(1).ToString());
        }

        transaction4.Commit();
    }

    // Roll back the first transaction
    transaction1.Rollback();
}

// CLEANUP
// Delete the TestSnapshot table and set
// ALLOW_SNAPSHOT_ISOLATION OFF
using (SqlConnection connection5 = new SqlConnection(connectionString))
{
    connection5.Open();
    SqlCommand command5 = connection5.CreateCommand();
    command5.CommandText = "DROP TABLE TestSnapshot";
    SqlCommand command6 = connection5.CreateCommand();
    command6.CommandText =
        "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION OFF";
    try
    {
        command5.ExecuteNonQuery();
        command6.ExecuteNonQuery();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
Console.WriteLine("Done!");
```

```
' Assumes GetConnectionString returns a valid connection string
' where pooling is turned off by setting Pooling=False;.
Dim connectionString As String = GetConnectionString()

Using connection1 As New SqlConnection(connectionString)
    ' Drop the TestSnapshot table if it exists
    connection1.Open()
    Dim command1 As SqlCommand = connection1.CreateCommand
    command1.CommandText = "IF EXISTS " & _
    "(SELECT * FROM sys.tables WHERE name=N'TestSnapshot') " _
      & "DROP TABLE TestSnapshot"
    Try
        command1.ExecuteNonQuery()
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try

    ' Enable SNAPSHOT isolation
    command1.CommandText = _
    "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION ON"
    command1.ExecuteNonQuery()
```

```
' Create a table named TestSnapshot and insert one row of data
command1.CommandText = _
"CREATE TABLE TestSnapshot (ID int primary key, valueCol int)"
command1.ExecuteNonQuery()
command1.CommandText = _
  "INSERT INTO TestSnapshot VALUES (1,1)"
command1.ExecuteNonQuery()

' Begin, but do not complete, a transaction to update the data
' with the Serializable isolation level, which locks the table
' pending the commit or rollback of the update. The original
' value in valueCol was 1, the proposed new value is 22.
Dim transaction1 As SqlTransaction = _
  connection1.BeginTransaction(IsolationLevel.Serializable)
command1.Transaction = transaction1
command1.CommandText = _
 "UPDATE TestSnapshot SET valueCol=22 WHERE ID=1"
command1.ExecuteNonQuery()

' Open a second connection to AdventureWorks
Dim connection2 As SqlConnection = New SqlConnection(connectionString)
Using connection2
    connection2.Open()

    ' Initiate a second transaction to read from TestSnapshot
    ' using Snapshot isolation. This will read the original
    ' value of 1 since transaction1 has not yet committed.
    Dim command2 As SqlCommand = connection2.CreateCommand()
    Dim transaction2 As SqlTransaction = _
      connection2.BeginTransaction(IsolationLevel.Snapshot)
    command2.Transaction = transaction2
    command2.CommandText = _
        "SELECT ID, valueCol FROM TestSnapshot"
    Dim reader2 As SqlDataReader = _
        command2.ExecuteReader()
    While reader2.Read()
        Console.WriteLine("Expected 1,1 Actual " _
          & reader2.GetValue(0).ToString() + "," _
          & reader2.GetValue(1).ToString())
    End While
    transaction2.Commit()
End Using

' Open a third connection to AdventureWorks and
' initiate a third transaction to read from TestSnapshot
' using the ReadCommitted isolation level. This transaction
' will not be able to view the data because of
' the locks placed on the table in transaction1
' and will time out after 4 seconds.
' You would see the same behavior with the
' RepeatableRead or Serializable isolation levels.
Dim connection3 As SqlConnection = New SqlConnection(connectionString)
Using connection3
    connection3.Open()
    Dim command3 As SqlCommand = connection3.CreateCommand()
    Dim transaction3 As SqlTransaction = _
        connection3.BeginTransaction(IsolationLevel.ReadCommitted)
    command3.Transaction = transaction3
    command3.CommandText = _
        "SELECT ID, valueCol FROM TestSnapshot"
    command3.CommandTimeout = 4

    Try
        Dim reader3 As SqlDataReader = command3.ExecuteReader()
        While reader3.Read()
            Console.WriteLine("You should never hit this.")
        End While
        transaction3.Commit()
```

```
        Catch ex As Exception
            Console.WriteLine("Expected timeout expired exception: " _
              & ex.Message)
            transaction3.Rollback()
        End Try
    End Using

    ' Open a fourth connection to AdventureWorks and
    ' initiate a fourth transaction to read from TestSnapshot
    ' using the ReadUncommitted isolation level. ReadUncommitted
    ' will not hit the table lock, and will allow a dirty read
    ' of the proposed new value 22. If the first transaction
    ' transaction rolls back, this value will never actually have
    ' existed in the database.
    Dim connection4 As SqlConnection = New SqlConnection(connectionString)
    Using connection4
        connection4.Open()
        Dim command4 As SqlCommand = connection4.CreateCommand()
        Dim transaction4 As SqlTransaction = _
          connection4.BeginTransaction(IsolationLevel.ReadUncommitted)
        command4.Transaction = transaction4
        command4.CommandText = _
            "SELECT ID, valueCol FROM TestSnapshot"
        Dim reader4 As SqlDataReader = _
            command4.ExecuteReader()
        While reader4.Read()
            Console.WriteLine("Expected 1,22 Actual " _
              & reader4.GetValue(0).ToString() _
              & "," + reader4.GetValue(1).ToString())
        End While
        transaction4.Commit()

        ' Rollback transaction1
        transaction1.Rollback()
    End Using
End Using

' CLEANUP
' Drop TestSnapshot table and set
' ALLOW_SNAPSHOT_ISOLATION OFF for AdventureWorks
Dim connection5 As New SqlConnection(connectionString)
Using connection5
    connection5.Open()
    Dim command5 As SqlCommand = connection5.CreateCommand()
    command5.CommandText = "DROP TABLE TestSnapshot"
    Dim command6 As SqlCommand = connection5.CreateCommand()
    command6.CommandText = _
   "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION OFF"
    Try
        command5.ExecuteNonQuery()
        command6.ExecuteNonQuery()
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try
End Using
```

Example

The following example demonstrates the behavior of snapshot isolation when data is being modified. The code performs the following actions:

- Connects to the **AdventureWorks** sample database and enables SNAPSHOT isolation.
- Creates a table named **TestSnapshotUpdate** and inserts three rows of sample data.
- Begins, but does not complete, sqlTransaction1 using SNAPSHOT isolation. Three rows of data are selected in the transaction.

- Creates a second **SqlConnection** to **AdventureWorks** and creates a second transaction using the READ COMMITTED isolation level that updates a value in one of the rows selected in sqlTransaction1.
- Commits sqlTransaction2.
- Returns to sqlTransaction1 and attempts to update the same row that sqlTransaction1 already committed. Error 3960 is raised, and sqlTransaction1 is rolled back automatically. The **SqlException.Number** and **SqlException.Message** are displayed in the Console window.
- Executes clean-up code to turn off snapshot isolation in **AdventureWorks** and delete the **TestSnapshotUpdate** table.

```
// Assumes GetConnectionString returns a valid connection string
// where pooling is turned off by setting Pooling=False;.
string connectionString = GetConnectionString();
using (SqlConnection connection1 = new SqlConnection(connectionString))
{
    connection1.Open();
    SqlCommand command1 = connection1.CreateCommand();

    // Enable Snapshot isolation in AdventureWorks
    command1.CommandText =
        "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION ON";
    try
    {
        command1.ExecuteNonQuery();
        Console.WriteLine(
            "Snapshot Isolation turned on in AdventureWorks.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("ALLOW_SNAPSHOT_ISOLATION ON failed: {0}", ex.Message);
    }
    // Create a table
    command1.CommandText =
        "IF EXISTS "
        + "(SELECT * FROM sys.tables "
        + "WHERE name=N'TestSnapshotUpdate')"
        + " DROP TABLE TestSnapshotUpdate";
    command1.ExecuteNonQuery();
    command1.CommandText =
        "CREATE TABLE TestSnapshotUpdate "
        + "(ID int primary key, CharCol nvarchar(100));";
    try
    {
        command1.ExecuteNonQuery();
        Console.WriteLine("TestSnapshotUpdate table created.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("CREATE TABLE failed: {0}", ex.Message);
    }
    // Insert some data
    command1.CommandText =
        "INSERT INTO TestSnapshotUpdate VALUES (1,N'abcdefg');"
        + "INSERT INTO TestSnapshotUpdate VALUES (2,N'hijklmn');"
        + "INSERT INTO TestSnapshotUpdate VALUES (3,N'opqrstuv');";
    try
    {
        command1.ExecuteNonQuery();
        Console.WriteLine("Data inserted TestSnapshotUpdate table.");
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
```

```
    // Begin, but do not complete, a transaction
    // using the Snapshot isolation level.
    SqlTransaction transaction1 = null;
    try
    {
        transaction1 = connection1.BeginTransaction(IsolationLevel.Snapshot);
        command1.CommandText =
            "SELECT * FROM TestSnapshotUpdate WHERE ID BETWEEN 1 AND 3";
        command1.Transaction = transaction1;
        command1.ExecuteNonQuery();
        Console.WriteLine("Snapshot transaction1 started.");

        // Open a second Connection/Transaction to update data
        // using ReadCommitted. This transaction should succeed.
        using (SqlConnection connection2 = new SqlConnection(connectionString))
        {
            connection2.Open();
            SqlCommand command2 = connection2.CreateCommand();
            command2.CommandText = "UPDATE TestSnapshotUpdate SET CharCol="
                + "N'New value from Connection2' WHERE ID=1";
            SqlTransaction transaction2 =
                connection2.BeginTransaction(IsolationLevel.ReadCommitted);
            command2.Transaction = transaction2;
            try
            {
                command2.ExecuteNonQuery();
                transaction2.Commit();
                Console.WriteLine(
                    "transaction2 has modified data and committed.");
            }
            catch (SqlException ex)
            {
                Console.WriteLine(ex.Message);
                transaction2.Rollback();
            }
            finally
            {
                transaction2.Dispose();
            }
        }

        // Now try to update a row in Connection1/Transaction1.
        // This transaction should fail because Transaction2
        // succeeded in modifying the data.
        command1.CommandText =
            "UPDATE TestSnapshotUpdate SET CharCol="
            + "N'New value from Connection1' WHERE ID=1";
        command1.Transaction = transaction1;
        command1.ExecuteNonQuery();
        transaction1.Commit();
        Console.WriteLine("You should never see this.");
    }
    catch (SqlException ex)
    {
        Console.WriteLine("Expected failure for transaction1:");
        Console.WriteLine("  {0}: {1}", ex.Number, ex.Message);
    }
    finally
    {
        transaction1.Dispose();
    }
}

// CLEANUP:
// Turn off Snapshot isolation and delete the table
using (SqlConnection connection3 = new SqlConnection(connectionString))
{
    connection3.Open();
```

```
    SqlCommand command3 = connection3.CreateCommand();
    command3.CommandText =
        "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION OFF";
    try
    {
        command3.ExecuteNonQuery();
        Console.WriteLine(
            "CLEANUP: Snapshot isolation turned off in AdventureWorks.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("CLEANUP FAILED: {0}", ex.Message);
    }
    command3.CommandText = "DROP TABLE TestSnapshotUpdate";
    try
    {
        command3.ExecuteNonQuery();
        Console.WriteLine("CLEANUP: TestSnapshotUpdate table deleted.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("CLEANUP FAILED: {0}", ex.Message);
    }
}
```

```
' Assumes GetConnectionString returns a valid connection string
' where pooling is turned off by setting Pooling=False;.
Dim connectionString As String = GetConnectionString()

Using connection1 As New SqlConnection(connectionString)
    ' Enable Snapshot isolation in AdventureWorks
    connection1.Open()
    Dim command1 As SqlCommand = connection1.CreateCommand
    command1.CommandText = _
   "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION ON;"
    Try
        command1.ExecuteNonQuery()
        Console.WriteLine( _
         "Snapshot Isolation turned on in AdventureWorks.")
    Catch ex As Exception
        Console.WriteLine("ALLOW_SNAPSHOT_ISOLATION failed: {0}", ex.Message)
    End Try

    ' Create a table
    command1.CommandText = _
      "IF EXISTS (SELECT * FROM sys.databases " _
      & "WHERE name=N'TestSnapshotUpdate') " _
      & "DROP TABLE TestSnapshotUpdate"
    command1.ExecuteNonQuery()
    command1.CommandText = _
      "CREATE TABLE TestSnapshotUpdate (ID int primary key, " _
      & "CharCol nvarchar(100));"
    Try
        command1.ExecuteNonQuery()
        Console.WriteLine("TestSnapshotUpdate table created.")
    Catch ex As Exception
        Console.WriteLine("CREATE TABLE failed: {0}", ex.Message)
    End Try

    ' Insert some data
    command1.CommandText = _
      "INSERT INTO TestSnapshotUpdate VALUES (1,N'abcdefg');" _
      & "INSERT INTO TestSnapshotUpdate VALUES (2,N'hijklmn');" _
      & "INSERT INTO TestSnapshotUpdate VALUES (3,N'opqrstuv');"
    Try
        command1.ExecuteNonQuery()
        Console.WriteLine("Data inserted TestSnapshotUpdate table.")
```

```
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try

    ' Begin, but do not complete, a transaction
    ' using the Snapshot isolation level
    Dim transaction1 As SqlTransaction = Nothing
    Try
        transaction1 = connection1.BeginTransaction(IsolationLevel.Snapshot)
        command1.CommandText = _
          "SELECT * FROM TestSnapshotUpdate WHERE ID " _
          & "BETWEEN 1 AND 3"
        command1.Transaction = transaction1
        command1.ExecuteNonQuery()
        Console.WriteLine("Snapshot transaction1 started.")

        ' Open a second Connection/Transaction to update data
        ' using ReadCommitted. This transaction should succeed.
        Dim connection2 As SqlConnection = New SqlConnection(connectionString)
        Using connection2
            connection2.Open()
            Dim command2 As SqlCommand = connection2.CreateCommand()
            command2.CommandText = "UPDATE TestSnapshotUpdate SET " _
              & "CharCol=N'New value from Connection2' WHERE ID=1"
            Dim transaction2 As SqlTransaction = _
              connection2.BeginTransaction(IsolationLevel.ReadCommitted)
            command2.Transaction = transaction2
            Try
                command2.ExecuteNonQuery()
                transaction2.Commit()
                Console.WriteLine( _
                  "transaction2 has modified data and committed.")
            Catch ex As SqlException
                Console.WriteLine(ex.Message)
                transaction2.Rollback()
            Finally
                transaction2.Dispose()
            End Try
        End Using

        ' Now try to update a row in Connection1/Transaction1.
        ' This transaction should fail because Transaction2
        ' succeeded in modifying the data.
        command1.CommandText = _
          "UPDATE TestSnapshotUpdate SET CharCol=" _
            & "N'New value from Connection1' WHERE ID=1"
        command1.Transaction = transaction1
        command1.ExecuteNonQuery()
        transaction1.Commit()
        Console.WriteLine("You should never see this.")

    Catch ex As SqlException
        Console.WriteLine("Expected failure for transaction1:")
        Console.WriteLine("  {0}: {1}", ex.Number, ex.Message)
    Finally
        transaction1.Dispose()
    End Try
End Using

' CLEANUP:
' Turn off Snapshot isolation and delete the table
Dim connection3 As New SqlConnection(connectionString)
Using connection3
    connection3.Open()
    Dim command3 As SqlCommand = connection3.CreateCommand()
    command3.CommandText = _
  "ALTER DATABASE AdventureWorks SET ALLOW_SNAPSHOT_ISOLATION OFF"
    Try
        command3.ExecuteNonQuery()
```

```
        Console.WriteLine( _
         "Snapshot isolation turned off in AdventureWorks.")
    Catch ex As Exception
        Console.WriteLine("CLEANUP FAILED: {0}", ex.Message)
    End Try

    command3.CommandText = "DROP TABLE TestSnapshotUpdate"
    Try
        command3.ExecuteNonQuery()
        Console.WriteLine("TestSnapshotUpdate table deleted.")
    Catch ex As Exception
        Console.WriteLine("CLEANUP FAILED: {0}", ex.Message)
    End Try
End Using
```

Using Lock Hints with Snapshot Isolation

In the previous example, the first transaction selects data, and a second transaction updates the data before the first transaction is able to complete, causing an update conflict when the first transaction tries to update the same row. You can reduce the chance of update conflicts in long-running snapshot transactions by supplying lock hints at the beginning of the transaction. The following SELECT statement uses the UPDLOCK hint to lock the selected rows:

```
SELECT * FROM TestSnapshotUpdate WITH (UPDLOCK)
  WHERE PriKey BETWEEN 1 AND 3
```

Using the UPDLOCK lock hint blocks any rows attempting to update the rows before the first transaction completes. This guarantees that the selected rows have no conflicts when they are updated later in the transaction. For more information, see Hints (Transact-SQL).

If your application has many conflicts, snapshot isolation may not be the best choice. Hints should only be used when really needed. Your application should not be designed so that it constantly relies on lock hints for its operation.

See also

- SQL Server and ADO.NET
- ADO.NET Overview
- Transaction Locking and Row Versioning Guide

SqlClient Support for High Availability, Disaster Recovery

4/26/2022 • 5 minutes to read • Edit Online

This topic discusses SqlClient support (added in .NET Framework 4.5) for high-availability, disaster recovery with the Always On features -- Always On availability groups (AGs) and Always On failover cluster instances (FCIs) with SQL Server 2012 or later. For more information about either Always On feature, see SQL Server Books Online.

You can now specify an availability group listener or the name of an FCI in the connection property. If a SqlClient application is connected to a database that fails over, the original connection is broken and the application must open a new connection to continue work after the failover.

If you are not connecting to an AG or FCI, and if multiple IP addresses are associated with a hostname, SqlClient will iterate sequentially through all IP addresses associated with DNS entry. This can be time consuming if the first IP address returned by DNS server is not bound to any network interface card (NIC). When connecting an FCI, or to the listener of an availability group, SqlClient attempts to establish connections to all IP addresses in parallel and if a connection attempt succeeds, the driver discards any pending connection attempts.

> **NOTE**
>
> Increasing connection timeout and implementing connection retry logic will increase the probability that an application will connect to an availability group. Also, because a connection can fail because of a failover, you should implement connection retry logic, retrying a failed connection until it reconnects.

The following connection properties were added to SqlClient in .NET Framework 4.5:

- `ApplicationIntent`
- `MultiSubnetFailover`

You can programmatically modify these connection string keywords with:

1. ApplicationIntent
2. MultiSubnetFailover

> **NOTE**
>
> Setting `MultiSubnetFailover` to `true` isn't required with .NET Framework 4.6.1 or later versions.

Connecting With MultiSubnetFailover

Always specify `MultiSubnetFailover=True` when connecting to the FCI or the listener of an AG. `MultiSubnetFailover` enables faster failover for all AGs and or FCIs in SQL Server 2012 or later and significantly reduces failover time for single and multi-subnet Always On topologies. During a multi-subnet failover, the client attempts connections in parallel. During a subnet failover, the client aggressively retries the TCP connection.

The `MultiSubnetFailover` connection property indicates that the application is using either an AG or FCI and that

SqlClient will try to connect to the database on the primary SQL Server instance by trying to connect to all the IP addresses. When `MultiSubnetFailover=True` is specified for a connection, the client retries TCP connection attempts faster than the operating system's default TCP retransmit intervals. This enables faster reconnection after failover of either an AG or FCI, and is applicable to both single- and multi-subnet AGs and FCIs.

For more information about connection string keywords in SqlClient, see ConnectionString.

Specifying `MultiSubnetFailover=True` when connecting to something other than an AG or FCI may result in a negative performance impact, and is not supported.

Use the following guidelines to connect to a server using one of the Always On features:

- Use the `MultiSubnetFailover` connection property when connecting to a single subnet or multi-subnet; it will improve performance for both.
- To connect to an AG, specify the listener of the availability group as the server in your connection string.
- Connecting to a SQL Server instance configured with more than 64 IP addresses will cause a connection failure.
- Behavior of an application that uses the `MultiSubnetFailover` connection property is not affected based on the type of authentication: SQL Server Authentication, Kerberos Authentication, or Windows Authentication.
- Increase the value of `Connect Timeout` to accommodate for failover time and reduce application connection retry attempts.
- Distributed transactions are not supported.

If read-only routing is not in effect, connecting to a secondary replica location will fail in the following situations:

1. If the secondary replica location is not configured to accept connections.
2. If an application uses `ApplicationIntent=ReadWrite` (discussed below) and the secondary replica location is configured for read-only access.

SqlDependency is not supported on read-only secondary replicas.

A connection will fail if a primary replica is configured to reject read-only workloads and the connection string contains `ApplicationIntent=ReadOnly`.

Upgrading to Use Multi-Subnet Clusters from Database Mirroring

A connection error (ArgumentException) will occur if the `MultiSubnetFailover` and `Failover Partner` connection keywords are present in the connection string, or if `MultiSubnetFailover=True` and a protocol other than TCP is used. An error (SqlException) will also occur if `MultiSubnetFailover` is used and the SQL Server returns a failover partner response indicating it is part of a database mirroring pair.

If you upgrade a SqlClient application that currently uses database mirroring to a multi-subnet scenario, you should remove the `Failover Partner` connection property and replace it with `MultiSubnetFailover` set to `True` and replace the server name in the connection string with an availability group listener. If a connection string uses `Failover Partner` and `MultiSubnetFailover=True`, the driver will generate an error. However, if a connection string uses `Failover Partner` and `MultiSubnetFailover=False` (or `ApplicationIntent=ReadWrite`), the application will use database mirroring.

The driver will return an error if database mirroring is used on the primary database in the AG, and if `MultiSubnetFailover=True` is used in the connection string that connects to a primary database instead of to an availability group listener.

Specifying Application Intent

When `ApplicationIntent=ReadOnly`, the client requests a read workload when connecting to an AlwaysOn enabled database. The server will enforce the intent at connection time and during a USE database statement but only to an Always On enabled database.

The `ApplicationIntent` keyword does not work with legacy, read-only databases.

A database can allow or disallow read workloads on the targeted AlwaysOn database. (This is done with the `ALLOW_CONNECTIONS` clause of the `PRIMARY_ROLE` and `SECONDARY_ROLE` Transact-SQL statements.)

The `ApplicationIntent` keyword is used to enable read-only routing.

Read-Only Routing

Read-only routing is a feature that can ensure the availability of a read only replica of a database. To enable read-only routing:

1. You must connect to an Always On Availability Group availability group listener.
2. The `ApplicationIntent` connection string keyword must be set to `ReadOnly`.
3. The Availability Group must be configured by the database administrator to enable read-only routing.

It is possible that multiple connections using read-only routing will not all connect to the same read-only replica. Changes in database synchronization or changes in the server's routing configuration can result in client connections to different read-only replicas. To ensure that all read-only requests connect to the same read-only replica, do not pass an availability group listener to the `Data Source` connection string keyword. Instead, specify the name of the read-only instance.

Read-only routing may take longer than connecting to the primary because read only routing first connects to the primary and then looks for the best available readable secondary. Because of this, you should increase your login timeout.

See also

- SQL Server Features and ADO.NET
- ADO.NET Overview

SqlClient Support for LocalDB

4/26/2022 • 2 minutes to read • Edit Online

This article discusses how to connect to a LocalDB database. LocalDB is a lightweight version of SQL Server.

Remarks

To summarize what you can do with LocalDB:

- Create and start LocalDB instances with sqllocaldb.exe or your app.config file.
- Use sqlcmd.exe to add and modify databases in a LocalDB instance. For example, `sqlcmd -S (localdb)\myinst`.
- Use the `AttachDBFilename` connection string keyword to add a database to your LocalDB instance. When using `AttachDBFilename`, if you do not specify the name of the database with the `Database` connection string keyword, the database will be removed from the LocalDB instance when the application closes.
- Specify a LocalDB instance in your connection string. For example, your instance name is `myInstance`, the connection string would include:

  `server=(localdb)\\myInstance`

`User Instance=True` is not allowed when connecting to a LocalDB database.

For information about installing LocalDB, see SQL Server Express LocalDB.

Programmatically Create a Named Instance

An application can create a named instance and specify a database as follows:

- Specify the LocalDB instances to create in the app.config file, as follows. The version number of the instance should be the same as the version number of your LocalDB installation.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section
    name="system.data.localdb"

type="System.Data.LocalDBConfigurationSection,System.Data,Version=4.0.0.0,Culture=neutral,PublicKeyTo
ken=b77a5c561934e089"/>
  </configSections>
  <system.data.localdb>
    <localdbinstances>
      <add name="myInstance" version="11.0" />
    </localdbinstances>
  </system.data.localdb>
</configuration>
```

- Specify the name of the instance using the `server` connection string keyword. The instance name specified in the `server` connection string keyword must match the name specified in the app.config file.
- Use the `AttachDBFilename` connection string keyword to specify the .MDF file.

See also

- SQL Server Features and ADO.NET
- ADO.NET Overview

LINQ to SQL

4/26/2022 • 2 minutes to read • Edit Online

LINQ to SQL is a component of .NET Framework version 3.5 that provides a run-time infrastructure for managing relational data as objects.

> **NOTE**
> Relational data appears as a collection of two-dimensional tables (*relations* or *flat files*), where common columns relate tables to each other. To use LINQ to SQL effectively, you must have some familiarity with the underlying principles of relational databases.

In LINQ to SQL, the data model of a relational database is mapped to an object model expressed in the programming language of the developer. When the application runs, LINQ to SQL translates into SQL the language-integrated queries in the object model and sends them to the database for execution. When the database returns the results, LINQ to SQL translates them back to objects that you can work with in your own programming language.

Developers using Visual Studio typically use the Object Relational Designer, which provides a user interface for implementing many of the features of LINQ to SQL.

The documentation that is included with this release of LINQ to SQL describes the basic building blocks, processes, and techniques you need for building LINQ to SQL applications. You can also search Microsoft Docs for specific issues, and you can participate in the LINQ Forum, where you can discuss more complex topics in detail with experts. Finally, the LINQ to SQL: .NET Language-Integrated Query for Relational Data white paper details LINQ to SQL technology, complete with Visual Basic and C# code examples.

In This Section

Getting Started
Provides a condensed overview of LINQ to SQL along with information about how to get started using LINQ to SQL.

Programming Guide
Provides steps for mapping, querying, updating, debugging, and similar tasks.

Reference
Provides reference information about several aspects of LINQ to SQL. Topics include SQL-CLR Type Mapping, Standard Query Operator Translation, and more.

Samples
Provides links to Visual Basic and C# samples.

Related Sections

Language-Integrated Query (LINQ) - C#
Provides overviews of LINQ technologies in C#.

Language-Integrated Query (LINQ) - Visual Basic
Provides overviews of LINQ technologies in Visual Basic.

LINQ

Describes LINQ technologies for Visual Basic users.

LINQ and ADO.NET

Links to the ADO.NET portal.

LINQ to SQL Walkthroughs

Lists walkthroughs available for LINQ to SQL.

Downloading Sample Databases

Describes how to download sample databases used in the documentation.

LinqDataSource Web Server Control Overview

Describes how the LinqDataSource control exposes Language-Integrated Query (LINQ) to Web developers through the ASP.NET data-source control architecture.