Issue Date: FoxTalk December 2000

Using ADO and Visual FoxPro 6.0 to Access SQL Server 7.0 for Primary Key Generation

Richard Aman
Ricka@Bellsouth.net

Microsoft ActiveX Data Objects (ADO) enable client applications to access and manipulate data from a data source through an OLE DB provider. ADO's primary benefits are ease of use, high speed, low memory overhead, and a small disk footprint. ADO supports key features for building client/server and Web-based applications. Whereas ODBC provides access to various databases, ADO provides access to various types of data collections, including, but not limited to, databases. Richard Aman explains.

Several fine articles have been written by various Visual FoxPro experts to show how to use ADO objects with Visual FoxPro 6.0 to access data from a variety of sources, such as SQL Server 7.0, Oracle8i, Access, and even VFP tables. Using ADO objects such as the ADO.Connection object and the ADO.RecordSet object, it's very easy to connect to the data sources and work with the data. But there are two more objects that make it very easy to work with stored procedures in the databases—the ADO.Command object and the ADO.Parameter object. With these two objects (and the ADO.Connection object), applications can execute stored procedures, pass parameters to the stored procedures, and receive data back from the stored procedures.

Accessing SQL Server 7.0

This article will demonstrate how to use the ADO.Command object and the ADO.Parameter object along with the ADO.Connection object to access a table and a stored procedure in a SQL Server 7.0 database. The table and stored procedure will be used to generate unique primary key values for use when adding records to tables in the database.

One of the more difficult decisions of application design is determining the set of values to be used for primary keys in the tables of the application's database. Primary keys are used to uniquely identify individual records in a table. In conjunction with foreign keys, which relate data between parent and child tables, the primary keys are the only way to tie all pertinent data together for a particular entity, such as a patient record, an order, an inventory item, and so on.

Because the primary key is so important to relational database design, selecting the appropriate type of value is very important for ensuring uniqueness among the data. Some of the types of primary keys that have been tried are phone number (which doesn't work for a household with more than one person), name and address combinations (which fails when the address is a large facility), Social Security number (which is supposed to be unique but has been found to have duplicates on the black market), or even record numbers (which are unique but change during inserts, deletes, and packing). According to database design experts, the normalized database should use small, arbitrary values that hold no meaning other than being uniquely generated for a particular entity. Automatic incrementing integers are ideally suited for this task.

SQL Server 7.0 contains a field type called Identity, which is similar to an auto-incrementing field in Access or a sequence in Oracle. The SQL Server 7.0 Identity type is supposed to guarantee uniqueness of field values. However, there have been reports of the Identity numbers getting out of sequence, and of errors with primary key uniqueness failing during inserts and updates to SQL Server 7.0 databases. There's also a performance issue with using Identity fields, especially when the newly assigned number must be known before it's used. To use the assigned value for a new record's Identity field, the record must be added to the database in order to allow the database to assign the number, and then the record's Identity field must be requeried to determine which ID was assigned to the new record. This means performing two accesses of the database for each new record that's added.

Since Visual FoxPro doesn't have any sort of auto-incrementing field like the Identity field in SQL Server 7.0, VFP developers usually devise some mechanism by which the primary key values are kept in a table of counters that contain the last used or next available number for a particular primary key for a particular entity. VFP developers create a function that finds the record for the entity (such as patient or invoice), increments the current value by a set amount, stores the new value, and returns the new value to the calling procedure for use as the primary key value.

This article will show how to implement the same mechanism for generating unique primary key values by using a table of counters and a stored procedure in a SQL Server 7.0 database. The main focus will be on the Visual FoxPro function to access a SQL Server 7.0 stored procedure. This in no way is a tutorial on SQL Server 7.0 administration or programming.

The samples in this article were developed and tested using ADO 2.5, available from several online locations including http://www.microsoft.com/Data, which also contains the SDK, Help files, other resources, and a handy tool called the Component Checker Tool, which is designed to help the developer determine installed version information and diagnose MDAC installation issues.

There are several values for the various properties of the objects used in this article. The values used in the following examples have been described as to their particular meanings. To see the complete list of the valid property values for each ADO object, please refer to the documentation that's contained in the ADO SDK.

In SQL Server 7.0

Create a table in the SQL Server 7.0 database to hold the numbers that are to be used for sequencing the primary key values. Create a table called IdKeys with two columns:

FieldName	Type	Length
KeyName	VarChar	30
KeyValue	Int	4

Populate the table with KeyName values and starting KeyValue values:

KeyName	KeyValue
PATIENT	17
ORDER	35
INVOICE	113

...

Create a stored procedure in SQL Server 7.0 to access the table, locate the record with the KeyName value passed in, increment the KeyValue value by 1, store the result back to the table, and return the result in an OUTPUT parameter originally passed into the procedure. The code for this procedure is contained in the file SP_GETNEXTKEY.SQL, which is available in the accompanying **Download file**.

```
 CREATE PROCEDURE sp_GetNextKey
  @RetVal int OUTPUT,
  @Name varchar(30)
AS
  UPDATE IdKeys
  SET IdKeyValue = IdKeyValue + 1,
       @RetVal = IdKeyValue + 1
  FROM IdKeys
  WHERE IdKeyName = @Name
```

In Visual FoxPro 6.0

Create a procedure to send parameters to SQL Server 7.0 and receive the results. The code for this procedure is contained in the file GETNEXTKEY.PRG, which is also included in the **Download file**.

```
 *-- Function to retrieve the next ID value from a
*-- counter field in SQL Server 7.0
*--
FUNCTION GetNextKey
LPARAMETERS tcEntity
*-- Define variables
LOCAL lnRetVal, ;
  loConnection, ;
  lcConnectString, ;
  loCommand, ;
  loParameter1, ;
  loParameter2
*-- Init variables
lnRetVal = 0
*-- Create a Connection object and
*-- set properties in the Connection object.
loConnection = CREATEOBJECT( "ADODB.Connection" )
```

The CursorLocation property of the Connection object tells the provider where to create the cursor—either a server-side cursor or a client-side cursor.

We'll use a value of 2 for a server-side cursor:

```
loConnection.CursorLocation = 2
```

The ConnectionString property is used to specify a data source by passing a detailed connection string that contains a series of *argument* = *value* statements separated by semicolons; it's identical to the connection string that's set up in an ODBC connector.

Be sure to replace "Server1" with your server name and "sa" with your user ID, and fill in your password for login:

```
lcConnectString  = "Provider=SQLOLEDB.1;" + ;
"Data Source=Server1;User ID=sa;Password= "
```

Open the connection to the SQL Server 7.0 database using the properties set previously:

```
loConnection.Open( lcConnectString )
```

Now that we've established a connection to our SQL Server 7.0 database, we can create and set properties in the Command object:

```
loCommand = CREATEOBJECT( "ADODB.Command" )
```

The CommandType property is used to assist ADO in the evaluation of the CommandText property. By specifically identifying the type of Command object, ADO can optimize the command text before sending it to the provider.

We'll use a value of 4 to tell ADO that we're sending the name of a stored procedure:

```
loCommand.CommandType = 4
```

The CommandText property contains the text of a command represented by a Command object. Usually this will be a SQL statement, but it can also be any other type of command statement that's recognized by the provider, such as a stored procedure call as we use in this example. If the CommandText property contains a SQL statement, it must be of the particular dialect or version supported by the provider's query processor.

We'll use sp_GetNextKey as the name of our stored procedure in SQL Server 7.0:

```
loCommand.CommandText = "sp_GetNextKey"
```

The ActiveConnection property of the Command object is used to indicate the Connection object over which the specified Command object will execute.

We'll use loConnection, which is the connection we opened earlier:

```
loCommand.ActiveConnection = loConnection
```

Now we'll create and append two Parameter objects to the Command object. Note that the type, size, direction, and order must match the parameters in the stored procedure.

The syntax for creating a Parameter object is:

```
CreateParameter(Name, Type, Direction, Size, Value)
```

The CreateParameter() method is used to create a new Parameter object with a specified name, type, direction, size, and value. Any values that are passed in the arguments are written to the corresponding Parameter properties.

All of the parameters for the CreateParameter() method are optional. We'll create our Parameter objects and then set the properties individually, rather than passing the property values in as part of the CreateParameter() method call.

This method doesn't automatically append the Parameter object to the Parameters collection of the Command object. This allows additional properties to be set whose values ADO will validate when the Parameter object is appended to the collection.

If a variable-length data type is specified in the Type argument, either a Size argument must be passed in or the Size property of the Parameter object must be set before appending it to the Parameters collection; otherwise, an error occurs.

If a numeric data type (numeric or decimal) is specified in the Type argument, the NumericScale and Precision properties must also be set.

The parameters we use in the CreateParameter() method are as follows:

- *Name*—Optional. A string value that contains the name of the Parameter object.
- *Type*—Optional. An Integer value that specifies the data type of the Parameter object.
- *Direction*—Optional. An Integer value that specifies the direction of the Parameter object.
- *Size*—Optional. A Long value that specifies the maximum length for the parameter value in characters or bytes.
- *Value*—Optional. A Variant that specifies the value for the Parameter object.

```
 *-- Create the OUTPUT Parameter object.
loParameter1 = loCommand.CreateParameter()
*-- Set the properties of the OUTPUT parameter.
loParameter1.Name = "KeyValue" && name we assign
loParameter1.Type = 3  && integer type parameter
loParameter1.Direction = 2 && output parameter
loParameter1.Size = 4 && maximum of 4 bytes long
loParameter1.Value = 0 && initialize the parameter to 0
*-- Append the OUTPUT parameter to the Command object.
loCommand.Parameters.Append( loParameter1 )
*-- Create the INPUT parameter object.
loParameter2 = loCommand.CreateParameter()
*-- Set the properties of the INPUT parameter.
loParameter2.Name = "KeyName"  && name we assign
loParameter2.Type = 201  && long varchar string type
loParameter2.Direction = 1  && input parameter
loParameter2.Size = 30 && maximum 30 characters
loParameter2.Value = tcEntity && our entity parameter
*-- Append the INPUT parameter to the Command object.
loCommand.Parameters.Append( loParameter2 )
*-- Execute the Command object to call the
*-- SQL Server 7.0 stored procedure.
loCommand.Execute()
```

Now comes the tricky part. Although everything in the Help files indicates that the value is returned directly, such as a function call would do, the result is actually contained in the value property of the Parameter object with the OUTPUT type specified (in our case, the KeyValue parameter).

Using this as lnRetVal = loCommand.Execute(), you'll get incorrect or missing results.

```
 *-- Retrieve the key value.
lnRetVal = loCommand.Parameters("KeyValue").Value
*-- Close and release the objects, and return the
*-- value to the caller.
RELEASE loCommand
loConnection.Close
RELEASE loConnection
RETURN( lnRetVal )
```

To get the next key value for an entity, execute this VFP procedure from within your code:

```
 LnPrimaryKey = GetNextKey( "PATIENT" )
```

This will call the stored procedure in SQL Server 7.0, which will update the PATIENT counter in the IdKeys table and return the result to VFP to be used in your own procedures.

Conclusion

In this article, we've examined some of the more common uses of ADO from a standpoint of using Visual FoxPro as a front end to a SQL Server 7.0 database. ADO is extremely fast (much faster than corresponding ODBC calls). Wrapping the functionality of OLE DB in ADO makes accessing disparate back-end databases quick and easy. Simply changing the

connection string of the Connection object (if all else is similar between the back ends) allows the application to function independently of the data source.

Once the developer has a basic understanding of SQL Server 7.0 stored procedures and how to access them with Visual FoxPro, this technology's uses in developing applications for various back ends are limitless.

The examples contained in this article use no error trapping. The focus has been on the usage of the ADO objects as they pertain to accessing data. ADO does contain an Error object, which can hold one or more errors depending on the ADO functionality being used. It's good programming practice to check the Error object after each call to ADO to handle any errors that might occur during data access with ADO.