

Managing an @@Identity Crisis

William Vaughn

Letting SQL Server generate Identity values can make your job harder if you don't know how to retrieve the new Identity values.

As some of you know, I spend a portion of my day trolling the newsgroups fishing for interesting topics. Nowadays I stay away from JET/Access questions—I'm no longer impartial enough to answer these folks without putting a knot in my stomach. I keep thinking, "You wouldn't be asking this question if you weren't using a toy database engine." But that's the fodder for another article. Each week there are usually several questions that discussed how to handle new Identity values. For example, once you add a new row to a database table that includes an Identity column, how can you tell what number the server assigned to the Identity column? This article discusses the first of two strategies that I explain in my books and classes. I call this first strategy "Post-INSERT fetching" as it executes an additional SELECT query that returns the newly created Identity value after the INSERT is executed. But there's more to handling Identity values than simply fetching the newly generated value. What about the values created on the client? How can one manage these values when you're creating parent-child relationships and need to have valid Identity values to manage inter-table relationships? We'll discuss that too.

The other technique, called "Bulk INSERT" or the "anvil salesman's paradigm" is covered in a later article. This technique that pre-builds child rows parallels schemes used for centuries to track orders and items using paper-based tracking systems. Basically, this approach pre-builds new child rows on the server, and updates these rows with current information as the need arises. The system-generated identity values are already in place so they don't have to be fetched when adding "new" rows.

What are Identity Columns?

Before we wade into the swamp of Identity details, let's clear up a few concepts for those not up to speed on the fundamentals. An Identity column is used to provide a unique Integer value that's guaranteed to be unique in the table *within the scope of the server*—and no further. That is, if you have several servers spread all over the world, there's no guarantee that the Identity values generated for the "CustomerOrders" table in Boston won't collide with the values generated for the same table in another identical database in Cleveland. This means that if you want a unique number that's guaranteed to be unique worldwide, you can't use an Identity without a further qualifier (such as a system ID), so you should consider use of a GUID UNIQUEIDENTIFIER instead¹. For the context of this article, let's just assume that we're working with a single DBMS server and don't care about replicating with another server's database.

How does SQL Server Manage Identity Values?

When you add a row to a table with an Identity column you don't include a value for the Identity column because, the DBMS server automatically adds an increment (usually 1) to the highest Identity value in the table and uses this value for the new row's Identity value. If you're using SQL Server, the value is also saved in a connection-global variable: @@IDENTITY². That's fine, but what happens when a row is deleted? Is that row's Identity value forever orphaned? Yep, unless you reseed the Identity (DBCC CHECKIDENT), deleted Identity values are lost. Identity values are also orphaned when a transaction is rolled back. This means that when you use Identity columns, you'll need to be prepared for gaps in the series. It also means that eventually, the Integer you're using will overflow, so it's important that you use an Integer large enough for your needs—now, and well into the future. An "integer" in SQL Server can identify about 2 billion rows while a "bigint" can identify 9,223,372,036,854,775,807 rows (that's a lot of rows). However, a "smallint" can only identify about 32 thousand rows. I actually had someone complain that he ran out of Identity values—they had used a "tinyint" which ran out after 255 rows. Sigh. I'm not going to delve into techniques to recover orphaned identity values—it's tough to do and over the years I've found it's not worth the trouble. Just make sure to define an integer wide enough to get you through the next century or corporate take-over when they re-write everything anyway.

How ADO.NET Handles Identity Values

ADO.NET has its own mechanism to handle client-side Identity values, because ADO.NET works with "disconnected" data and does not expect to have live access to the "real" server-side data table. This means that

¹ GUID identifiers are supported in SQL Server and other full-featured DBMS systems, but not Access/JET.

² You can use similar techniques to retrieve server-generated values from other non-SQL Server databases as well. MySQL developers can use the "LAST_INSERT_ID()". Oracle developers can use "CURRVAL" to retrieve the last value generated for a sequence on the connection.

as you add rows to a DataTable object on your client, the Identity value generated *locally* by ADO.NET won't have any bearing on the Identity values of existing rows in the database *or* on the rows in the local DataTable. Huh? How can that work? Well, ADO.NET does not make any effort to test the Identity values it generates against any existing Identity values in the client-side disconnected table. It sets the new Identity values based on the AutoIncrementSeed and AutoIncrementStep. This means that if your existing DataTable has identity values ranging from 1 to 10 and you set the AutoIncrementSeed value to 10, ADO.NET does what it's told—it starts at 10 and you'll end up with two rows with the same Identity value (not good). These “autoincrement” properties can be set before you add any rows to the DataTable—afterwards, it does not seem to matter. The demonstration application included with this article illustrates this behavior.

“Tricking” ADO.NET

When setting a client-side Identity value, the trick is to set a value that the server-side database table is *not* using or not likely to be used by rows being added by other clients. That way, any new rows won't collide with existing rows in the current DataTable. Since server-side Identity values are usually positive integers, you have only one other set of numbers to use on the client—negative numbers. ADO.NET is ready to handle this contingency—set both the *AutoIncrementSeed* and the *AutoIncrementStep* to -1. This way each new row is born with an Identity unique to the client. This also means that a row in a parent-child relationship can easily identify its relations before the server assigns their “real” identity when you execute the INSERT query. These negative numbers are used to interrelate parents with their children until ADO.NET inserts the new rows into the server-side database with the Update method. Remember, the INSERT statements you or ADO.NET generate (via the CommandBuilder) *don't* include the client-side Identity values—they're only needed to identify client-side rows and relationships.

Parent-child relationships

Once you create a new parent row (for example, a row in the “Customer” table) and let ADO.NET generated its new Identity value, you can create as many child rows (for example, “Orders”) as necessary and safely use the parent's ADO.NET-generated Identity value as a foreign key (so the child row is tied back to the correct parent). Yes, ADO.NET knows how to handle these relationships correctly when it comes time to post these new rows to the server. When you execute the Update method, ADO.NET executes the INSERT for the parent row first and then all associated child rows. If you setup your InsertCommand correctly, the server-generated Identity value is propagated to the child foreign key value. Listing 1 shows the “magic” part of the Update button click event handler that executes the Update method several times in the right sequence. The first time Update is invoked, I add any new and update any existing parent rows—but rows marked for deletion are left in the DataSet. The second call to Update I make all changes to the child table (any adds, changes or deletes). This way children are added after their parents but deleted only after the parents are deleted. The last update deletes any parent rows marked for removal so the children are deleted first (before their parents).

```
Try
    ' Add parents first, then children
    ' Delete children first, then parents
    ' Use the Select method to return an array of rows to be updated or added
daParent.Update(ds.Tables(eTbl.Parent).Select("", "", _
    DataViewRowState.Added Or DataViewRowState.ModifiedCurrent))
daChild.Update(ds.Tables(eTbl.Child)) ' Add, change or delete children
daParent.Update(ds.Tables(eTbl.Parent)) ' Delete any remaining parents
```

Listing 1 – Correct Update method sequencing

Tip: Nope, you don't see a call to the AcceptChanges method in this code—it's not necessary as it's called automatically by the Update method after it posts its changes to the database.

Retrieving the New Identity Values

The real problems come when you want to find out what Identity values have been generated by the server-side DBMS engine. Unfortunately, ADO.NET does nothing on its own to help. Visual Studio's DataAdapter Configuration Wizard (DACW) can help generate additional SQL for your InsertCommand to retrieve the new row's Identity but the CommandBuilder is clueless in this regard so it's no help at all. Since I usually roll my own action query SQL, neither of these approaches help—at least not much.

One approach would be to simply re-query. That is, rerun the entire SelectCommand to rebuild/refresh the DataSet, but that's overkill. All you need is the new row—not all the rows in the DataTable's rowset. Taking the DACW's lead, we can either leverage the code it generates or add equivalent code to our own INSERT SQL command to be executed by the DataAdapter object's InsertCommand. To get the DACW to generate the extra

SQL to return the new Identity, you don't have to do anything except run the DACW wizard—the default “Advanced” setting does this for you.

Refresh the DataSet

Adds a Select statement after Insert and Update statements to retrieve identity column values, default values, and other values calculated by the database.

Figure 1. (DACW.tif) DataAdapter Configuration Wizard “Advanced” option.

An example of code generated by the DACW to perform the INSERT and retrieve the new Identity value is shown in Listing 2. Drill down into the code region generated by the Windows Form Designer to find it after you run the DACW.

```
#Region " Windows Form Designer generated code "  
...  
'SqlInsertCommand1  
Me.SqlInsertCommand1.CommandText = "INSERT INTO TestInsert(Name, State) _"  
& "VALUES (@Name, @State);" _  
& " SELECT ID, Name, Stat FROM TestInsert WHERE (ID = @@IDENTITY)"  
...'
```

Listing 2: InsertCommand as generated by the DACW

Note that the DACW added another SQL SELECT statement after the INSERT to return all values from the new row—including the new Identity value.

Problems with @@IDENTITY

Unfortunately, the DACW generated-code assumes that your database does not expect any triggers to fire when the INSERT is executed. If a trigger does fire and if that trigger adds another row to a table, the @@IDENTITY global variable would be set to point to that new Identity value—not the one your INSERT generated. This makes the DACW-generated code work for simple situations, but not when your database gets more sophisticated. Hopefully, that won't happen until after you retire. The solution? Instead of using the error-prone @@IDENTITY global variable, your code should use the new SCOPE_IDENTITY() function. It returns the inner-most Identity value and is unaffected by other INSERT operations done in other code scopes such as triggers. The code should look like Listing 3—at least with SQL Server 2000 or later:

```
'SqlInsertCommand1  
Me.SqlInsertCommand1.CommandText = "INSERT INTO TestInsert(Name, State) _"  
& "VALUES (@Name, @State);" _  
& " SELECT ID, Name, Stat FROM TestInsert WHERE (ID = SCOPE_IDENTITY())"  
...'
```

Listing 3: Corrected InsertCommand

Too bad the DACW does not give you this option.

Access/JET Issues

Unlike SQL Server and other more powerful DBMS systems, the JET database engine (as used in Access and other smaller applications) cannot execute multiple statements in a single batch. This means you'll have to take another course—execute another query using the DataAdapter RowUpdated event. Sure, JET supports @@IDENTITY (in Access 2000 and later) so you can submit a “SELECT @@IDENTITY” query in the event handler to capture the new Identity value. You'll have to copy the value yourself into the Identity column in your DataRow—ADO.NET won't do it for you. Be sure to call the DataAdapter AcceptChanges method after setting the value to keep from confusing ADO.NET into thinking that this was a user-generated change. Listing 3 illustrates some pseudo code to show how to create an event handler and capture the new row's Identity value.

```
Dim da As OleDbDataAdapter  
Dim cn As OleDbConnection  
Dim cmdGetIdentity As New OleDbCommand("SELECT @@IDENTITY", cn)
```

Declare your DataAdapter, and add an event handler for the RowUpdated event.

```
AddHandler da.RowUpdated, AddressOf RowUpdatedSetIdentity  
  
' Trap RowUpdated event  
Private Sub RowUpdatedSetIdentity(ByVal sender As Object, ByVal e As OleDbRowUpdatedEventArgs)  
If e.Status = UpdateStatus.Continue AndAlso _  
    e.StatementType = StatementType.Insert Then ' If this is an INSERT operation...
```

```

' Execute the post-update query to fetch new @@Identity
  e.Row("ID") = CInt(cmdGetIdentity.ExecuteScalar)
  e.Row.AcceptChanges()
End If

```

Listing 3: Capturing an Identity value in the RowUpdated event.

An important point here is that JET manages the @@IDENTITY value in the connection state. That is, each connection manages the new Identity value on its own which should prevent Identity collisions.

How ADO.NET Manages the INSERT Resultsets

Without going into gory detail, when you use the SQL Server double-query method described above, ADO.NET looks for these additional resultsets and posts the changes to the DataRow—even if they come from OUTPUT parameters. The UpdatedRowSource property controls this behavior. The result? Well, your DataRow object's Identity column is automatically set to the new server-side value. If you roll your own InsertCommand, you'll want to mimic this approach if you expect ADO.NET to handle the Identity values for you.

A code example

I wrote a sample application to illustrate the points in this article. It also addresses several other issues that might also affect how you approach Identity issues. The sample retrieves rows from two tables: TestInsertParent and TestInsertChild. The tables and their relationships are very simple to make coding easy. The PID column (ParentID) is a unique key for the parent table and a foreign key for the child. I hard-code this relationship and setup a Constraint to get ADO.NET to handle the cascading delete issue. That is, when I delete a parent row, I expect ADO.NET to also delete the child rows. To enforce this, when I create the tables (the code to do so is part of the program), I make sure the server-side constraints are also setup to prevent referential integrity problems. The DataGrid is used to display the initial rowset of parent rows. When the user clicks on the "+" sign in the DataGrid to drill down into the child rows, the sample executes a quick parameter query to return any related children. No, I don't believe in returning all parents and all child rows—even if there are only five rows of each. Sure, ADO.NET can do this, but this approach won't work particularly well if you have 10,000 parents and 50,000 children. Yes, it would be a good idea to include a parameter query when fetching the parent rows too. The code hard-codes the action commands used to maintain the parent and child tables. These six Command objects were initially created with the DataAdapter Configuration Wizard—but hand refined to make them more efficient and easier to maintain.

```

Dim cn As SqlConnection
Dim daParent, daChild As SqlDataAdapter
Dim ds As New DataSet()
Dim intPIDSelected As Integer = 0
Enum eTbl
    Parent
    Child
End Enum
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Try
        ' Trap problems
        ' If you don't have the Biblio DB, change this to your own test DB (like Pubs)
        ' The application File menu has an entry that creates the test tables.
        cn = New SqlConnection("server=demoserver;database=biblio;integrated security=sspi")
        daParent = New SqlDataAdapter("SELECT PID, Name, State FROM TestInsertParent", cn)
        daChild = New SqlDataAdapter("SELECT CID, PID, ChildName, ChildAge" _
            & " FROM TestInsertChild WHERE PID = @PIDWanted", cn)
        daChild.SelectCommand.Parameters.Add("@PIDWanted", SqlDbType.Int)
        ' Set MissingSchemaAction to make sure multiple Fills don't add,
        ' but update data in DataTable
        daChild.MissingSchemaAction = MissingSchemaAction.AddWithKey
        GenerateCommands()
        ' Build the Action Commands
    Catch ex As Exception
        MsgBox(ex.ToString)
    End Try
End Sub
Private Sub CreateRelations()
    ' Create inter-DataTable Relation objects between the Parent and Child tables
    Dim colParent As DataColumn = ds.Tables(eTbl.Parent).Columns("PID")
    Dim colChild As DataColumn = ds.Tables(eTbl.Child).Columns("PID")
    ' Create Relation along with Constraint
    Dim daRel As New DataRelation("ParentToChildRelation", colParent, colChild, True)
    ds.Relations.Add(daRel)
End Sub
Private Sub btnUpdate_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnUpdate.Click

```

```

Try
    ' Add parents first, then children
    ' Delete children first, then parents
    ' Use the Select method to return an array of rows to be updated or added
    daParent.Update(ds.Tables(eTbl.Parent).Select("", "", _
        DataViewRowState.Added Or DataViewRowState.ModifiedCurrent))
    daChild.Update(ds.Tables(eTbl.Child)) ' Add, change or delete children
    daParent.Update(ds.Tables(eTbl.Parent)) ' Delete any remaining parents
Catch exsql As SqlException
    If exsql.Number = 547 Then
        MsgBox("You must first delete child rows ... ")
    Else
        MsgBox(exsql.ToString)
    End If
Catch ex As Exception
    MsgBox(ex.ToString)
End Try
End Sub
Private Sub btnSetAutoIncrement_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnSetAutoIncrement.Click
Try
    ' Set Autoincrement, seed, and step
    With ds.Tables(eTbl.Parent).Columns("PID")
        .AutoIncrement = True
        .AutoIncrementSeed = CInt(txtAutoIncrementSeed.Text)
        .AutoIncrementStep = CInt(txtAutoIncrementStep.Text)
    End With
Catch ex As Exception
End Try
End Sub
Private Sub GenerateCommands()
    ' These commands were originally generated by the DACW and tuned to remove
    ' some"imperfections" as discussed in the text of the article.
    daParent.InsertCommand = New SqlCommand()
With daParent.InsertCommand
    .CommandText = "INSERT INTO TestInsertParent(Name, State) VALUES (@Name, @State); " _
    & " SELECT PID, Name, State FROM TestInsertParent WHERE (PID = SCOPE_IDENTITY())"
    .Connection = cn
    ' Set Name, datatype, size and source column.
    .Parameters.Add("@Name", System.Data.SqlDbType.VarChar, 50, "Name")
    .Parameters.Add("@State", System.Data.SqlDbType.VarChar, 50, "State")
End With
daParent.UpdateCommand = New SqlCommand()
With daParent.UpdateCommand
    .CommandText = "UPDATE TestInsert SET Name = @Name, State = @State " _
    & "WHERE (PID = @Original_PID) AND (Name = @Original_Name) AND " _
    & "(State = @Original_State); " _
    & "SELECT PID, Name, State FROM TestInsertParent WHERE (PID = @PID)"
    .Connection = cn
    .Parameters.Add("@Name", System.Data.SqlDbType.VarChar, 50, "Name")
    .Parameters.Add("@State", System.Data.SqlDbType.VarChar, 50, "State")
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_PID", _
        System.Data.SqlDbType.Int, 4, System.Data.ParameterDirection.Input, False, _
        CType(0, Byte), CType(0, Byte), "PID", System.Data.DataRowVersion.Original, Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Name", _
        System.Data.SqlDbType.VarChar, 50, System.Data.ParameterDirection.Input, _
        False, CType(0, Byte), CType(0, Byte), "Name", _
        System.Data.DataRowVersion.Original, Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_State", _
        System.Data.SqlDbType.VarChar, 50, System.Data.ParameterDirection.Input, _
        False, CType(0, Byte), CType(0, Byte), "State", System.Data.DataRowVersion.Original,
        Nothing))
    .Parameters.Add("@PID", System.Data.SqlDbType.Int, 4, "PID")
End With
daParent.DeleteCommand = New SqlCommand()
With daParent.DeleteCommand
    .CommandText = "DELETE FROM TestInsertParent WHERE (PID = @Original_PID) " _
    & "AND (Name = @Original_Name) AND (State = @Original_State)"
    .Connection = cn
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_PID", _
        System.Data.SqlDbType.Int, 4, System.Data.ParameterDirection.Input, _
        False, CType(0, Byte), CType(0, Byte), "PID", System.Data.DataRowVersion.Original, Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_Name", _
        System.Data.SqlDbType.VarChar, 50, System.Data.ParameterDirection.Input, _
        False, CType(0, Byte), CType(0, Byte), "Name", System.Data.DataRowVersion.Original,
        Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_State", _

```

```

        System.Data.SqlDbType.VarChar, 50, System.Data.ParameterDirection.Input, _
        False, CType(0, Byte), CType(0, Byte), "State", System.Data.DataRowVersion.Original,
Nothing))
End With
' Generate action Commands for the Child DataAdapter
daChild.InsertCommand = New SqlCommand()
With daChild.InsertCommand
    .CommandText = "INSERT INTO TestInsertChild(PID, ChildName, ChildAge) " _
    & " VALUES (@PID, @ChildName, @ChildAge); " _
    & " SELECT CID, PID, ChildName, ChildAge " _
    & " FROM TestInsertChild WHERE (CID = SCOPE_IDENTITY())"
    .Connection = cn
    ' Set Name, datatype, size and source column.
    .Parameters.Add("@PID", System.Data.SqlDbType.Int, 2, "PID")
    .Parameters.Add("@ChildName", System.Data.SqlDbType.VarChar, 50, "ChildName")
    .Parameters.Add("@ChildAge", System.Data.SqlDbType.TinyInt, 1, "ChildAge")
End With
daChild.UpdateCommand = New SqlCommand()
With daChild.UpdateCommand
    .CommandText = "UPDATE TestInsert SET PID = @PID, Name = @ChildName, ChildAge = @ChildAge
" _
    & "WHERE (CID = @Original_CID) AND PID = @Original_PID) AND (ChildName =
@Original_Name) AND " _
    & "(ChildAge = @ChildAge); " _
    & "SELECT CID, PID, ChildName, ChildAge FROM TestInsertChild WHERE (CID = @CID)"
    .Connection = cn
    .Parameters.Add("@PID", System.Data.SqlDbType.Int, 2, "PID")
    .Parameters.Add("@ChildName", System.Data.SqlDbType.VarChar, 50, "ChildName")
    .Parameters.Add("@ChildAge", System.Data.SqlDbType.TinyInt, 1, "ChildAge")
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_CID", _
System.Data.SqlDbType.Int, 4, System.Data.ParameterDirection.Input, False, _
CType(0, Byte), CType(0, Byte), "CID", System.Data.DataRowVersion.Original, Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_PID", _
System.Data.SqlDbType.Int, 4, System.Data.ParameterDirection.Input, False, _
CType(0, Byte), CType(0, Byte), "PID", System.Data.DataRowVersion.Original, Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_ChildName", _
System.Data.SqlDbType.VarChar, 50, System.Data.ParameterDirection.Input, _
False, CType(0, Byte), CType(0, Byte), "ChildName", System.Data.DataRowVersion.Original,
Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_ChildAge", _
System.Data.SqlDbType.TinyInt, 1, System.Data.ParameterDirection.Input, _
False, CType(0, Byte), CType(0, Byte), "ChildAge", System.Data.DataRowVersion.Original,
Nothing))
    .Parameters.Add("@CID", System.Data.SqlDbType.Int, 4, "CID")
End With
daChild.DeleteCommand = New SqlCommand()
With daChild.DeleteCommand
    .CommandText = "DELETE FROM TestInsertChild WHERE (CID = @Original_CID) " _
    & "AND (PID = @Original_PID) AND (ChildName = @Original_ChildName) AND (ChildAge =
@Original_ChildAge)"
    .Connection = cn
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_CID", _
System.Data.SqlDbType.Int, 4, System.Data.ParameterDirection.Input, _
False, CType(0, Byte), CType(0, Byte), "CID", System.Data.DataRowVersion.Original, Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_PID", _
System.Data.SqlDbType.Int, 4, System.Data.ParameterDirection.Input, False, _
CType(0, Byte), CType(0, Byte), "PID", System.Data.DataRowVersion.Original, Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_ChildName", _
System.Data.SqlDbType.VarChar, 50, System.Data.ParameterDirection.Input, _
False, CType(0, Byte), CType(0, Byte), "ChildName", System.Data.DataRowVersion.Original,
Nothing))
    .Parameters.Add(New System.Data.SqlClient.SqlParameter("@Original_ChildAge", _
System.Data.SqlDbType.TinyInt, 1, System.Data.ParameterDirection.Input, _
False, CType(0, Byte), CType(0, Byte), "ChildAge", System.Data.DataRowVersion.Original,
Nothing))
End With
End Sub
Private Sub mnuCreateTables_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles mnuCreateTables.Click
' This routine is used to create the test tables used in this example.
Dim strMakeTable As New StringBuilder()
Try
    With strMakeTable
        .Append("IF NOT EXISTS(SELECT * FROM Sysobjects WHERE Name = 'TestInsertParent')")
        .Append("BEGIN CREATE TABLE [TestInsertParent] (")
        .Append(" [PID] [int] IDENTITY (1, 1) NOT NULL , ")
        .Append(" [Name] [varchar] (50) NOT NULL , ")
    End With

```

```

.Append(" [State] [varchar] (50) NOT NULL , ")
.Append(" CONSTRAINT [PK_TestInsertParent] PRIMARY KEY CLUSTERED ")
.Append(" ( [PID] ) ON [PRIMARY] ) ON [PRIMARY] END ")

' Create Child table with PK/FK relationship to Parent
.Append("IF NOT EXISTS(SELECT * FROM Sysobjects WHERE Name = 'TestInsertChild')")
.Append("BEGIN CREATE TABLE [TestInsertChild] (")
.Append(" [CID] [int] IDENTITY (1, 1) NOT NULL , ")
.Append(" [PID] [int] NOT NULL , ")
.Append(" [ChildName] [varchar] (50) NOT NULL , ")
.Append(" [ChildAge] [tinyint] NOT NULL , ")
.Append(" CONSTRAINT [PK_TestInsertChild] PRIMARY KEY CLUSTERED ")
.Append(" ( [CID] ) ON [PRIMARY], ")
.Append(" CONSTRAINT [FK_TestInsertChild_TestInsertParent] ")
.Append(" FOREIGN KEY([PID]) REFERENCES [TestInsertParent] ([PID])) ")
.Append(" ON [PRIMARY] ")
.Append(" END ")
    End With
    cn.Open()
    Dim cmdMakeTable As New SqlCommand(strMakeTable.ToString, cn)
    cmdMakeTable.ExecuteNonQuery()
    MsgBox("TestInsert Table created...")
Catch ex As Exception
    MsgBox(ex.ToString)
Finally
    cn.Close()
End Try
End Sub
Private Sub btnQuery_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles btnQuery.Click
Try
    ds.Clear() ' Clear out any previous contents of the DataSet (Parent and Child tables
cleared)
    daParent.Fill(ds, "TestInsertParent") ' Fill the Parent DataSet
    If daParent.TableMappings.Count = 0 Then
daParent.TableMappings.Add("Table", "TestInsertParent") ' Map logical to physical
    End If
    dgDisplay.DataSource = ds.Tables(eTbl.Parent) ' Bind Parent Ds to the DataGrid
    ' dgDisplay.DataMember = "TestInsertParent"
    btnSetAutoIncrement.PerformClick() ' Set the AutoIncrement values
    ' Fetch the Child table data based on the selected Parent
    If ds.Tables(eTbl.Parent).Rows.Count = 0 Then
MsgBox("Query did not return any rows. You can add new rows to the Parent grid")
    End If
    FetchChildRows()
    If ds.Relations.Count = 0 Then
CreateRelations()
    End If
Catch exSQL As SQLException
    If exSQL.Number = 208 Then
MsgBox("Did you create the TestInsert table(s)? See the File menu.")
    Else
MsgBox("Unexpected SQLException" & exSQL.ToString)
    End If
Catch ex As Exception
    MsgBox(ex.ToString)
End Try
End Sub
Private Sub dgDisplay_Navigate(ByVal sender As Object, ByVal ne As
System.Windows.Forms.NavigateEventArgs) Handles dgDisplay.Navigate
    btnUpdate.Enabled = ds.HasChanges
    If ne.Forward Then
        FetchChildRows()
    End If
End Sub
Private Sub FetchChildRows()
Try
    ' Extract PID from current row on DataGrid
    intPIDSelected = CInt(ds.Tables(eTbl.Parent).Rows(dgDisplay.CurrentRowIndex)("PID"))
    daChild.SelectCommand.Parameters(0).Value = intPIDSelected ' Search for PID's children
    daChild.Fill(ds, "TestInsertChild") ' Fill the Child DataSet from the database
    ' dgDisplay.DataSource = ds.Tables(eTbl.Parent) ' Rebind
    dgDisplay.Update()
    ' Map logical Child table to physical table
    If daChild.TableMappings.Count = 1 Then daChild.TableMappings.Add("Table1",
"TestInsertChild")
Catch ex As Exception

```

```

        MsgBox(ex.ToString)
    End Try
End Sub

Private Sub dgDisplay_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles dgDisplay.KeyDown
    ' If the user clicks Delete in the grid-
    ' be sure to fetch any child rows associated with this Parent
    ' to ensure that they get deleted before the parent.
    If e.KeyValue = 46 Then ' Delete key
        FetchChildRows()' Make sure that the child rows are included when Parent is deleted
    End If
End Sub

Private Sub mnuFileExit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles mnuFileExit.Click
    If ds.HasChanges Then
        Dim resp As MsgBoxResult = MsgBox("Do you want to save the changes made to your data?",
MsgBoxStyle.YesNo Or MsgBoxStyle.Question, "Uncommitted data")
        If resp = MsgBoxResult.Yes Then
            btnUpdate.PerformClick()
        End If
    End If
End Sub
End Class

```

Listing 4: Illustrate AutoIncrement effect on newly added rows.

William (Bill) Vaughn is president of Beta V Corporation based in Redmond, Washington not far from the Microsoft campus. He provides training and consulting services to clients around the globe, specializing in Visual Basic and SQL Server data access architecture and best practices. William's latest books are "ADO.NET and ADO Examples and Best Practices for Visual Basic Programmers--2nd Edition" and the C# version "ADO.NET Examples and Best Practices for C# Programmers". Both are available from Apress. William is also the author of many articles and training courses and is a top-rated speaker at several international computer conferences.

url: <http://www.betav.com>

email: billva@nwlinc.com