

.net, code

# Doing the Impossible with ADO.NET: Part 2

*William Vaughn*

**The said it couldn't be done - but they were wrong. Yes, you *can* create pessimistic locks on rows, pages, or even entire tables with SQL Server and ADO.NET.**

As I discussed last month, ADO.NET seemed to have lopped off most of the server-side functionality we had grown accustomed to under classic ADO - server-side cursors and pessimistic locks, for example. What's the rationale behind this move? Well, after having discussed this strategy with a couple of Product Unit Managers and developers at Microsoft, I can better understand their reasoning, but I'm still not sure I agree with it. The new "ADO" is designed to be lighter, leaner, and faster than the old and to discourage development techniques that don't make sense or hindered scalability or performance—especially in Web-centric designs. The problem is (at least as I see it), is that there are still plenty of developers out there creating "traditional" Windows forms applications that can use these features.

Last month I discussed how to use ADO.NET to create server-side cursors. This month I illustrate how to construct and manage pessimistic locks. But before we get started, let me state flatly that pessimistic locks are *not* a good idea for most applications and definitely not for highly interactive Web applications where you can't maintain server state on a connection basis. This should satiate those folks at Microsoft that cringe whenever I mention pessimistic locking. However, if you do *really* need pessimistic locks, this article shows you how to create, manage, and release them in relative safety.

## What are pessimistic locks?

Pessimistic locks are designed to permit your application to block changes by other applications (or even your own application) to one or more rows, pages, or tables for an indeterminate length of time. They call these "pessimistic" locks because it's assumed that it's likely that some other application will attempt to access the row(s) during the lifetime of the lock. The name has nothing to do with the fact that Republicans are holding both houses of Congress. Many developers just starting out choose this type of lock to avoid writing additional code to handle data "collisions"—when more than one user changes (or tries to change) the same data row at the same time. Pessimistic locks do not permit these changes to take place in the first place by locking out changes by other processes. Yes, this means that the "other process" might be created by your own application.

On the other hand, "optimistic" locking assumes that no other users will attempt to update your rowset's membership at the same time—or at least, they're not supposed to. In this case, the server makes no attempt to block changes to rows your application (or any other application) has already read. If multiple users change the same row after another user has read it, it's up to your code to detect the collision and settle the argument about which version should prevail. ADO.NET directly supports (and encourages) use of optimistic locking.

Pessimistic locks make sense for a very narrow set of circumstances--when a process (your application or a middle-tier component), needs to perform some operation that cannot permit the data row(s) in question to change while the operation takes place. Pessimistic locks should *not* be used to hold a row's state while the user (a human) decides what to do. That is, unless you have also included a mechanism to prevent the application from holding the lock indefinitely.

## Managing pessimistic locks

When you successfully establish a pessimistic lock on a selected row, other applications can't update that row until the lock is released. That does *not* mean, however, that other applications can't establish overlapping locks. That's because when a row, page or table is exclusively locked (which is the type of lock used when your application is granted a pessimistic lock), any other application's attempt to *change* the row is blocked by the server—the other application can still request a lock over the same row(s). The server holds all update requests for locked rows in a queue and waits until the lock(s) are released before performing the update. If the locks aren't released in *CommandTimeout* seconds, the updating application throws an exception indicating that the operation timed out. The default timeout is 30 seconds—an eon for SQL Server.

All locks are maintained by and on the server (or server instance) which is responsible for tracking all locked rows, pages, extents, and tables. This adds measurable processing and resource overhead to the server as it has to perform several additional tasks. These include verifying that any change requests do not infringe on locked rows, blocking processes that attempt to violate locks, releasing locks when requested, reactivating processes when locks are released, and returning exceptions when blocked processes time out.

## Pessimistic locking scope

It's easy to see how locking a row might only affect other users who try to change the same row. However, it's also easy to broaden the scope of the exclusive lock to include all of the rows on a page or all of the pages in an extent (eight pages in SQL Server) or all of the extents in a table. The scope of the exclusive lock is a function of how many rows qualify for your rowset which is determined by the query's WHERE clause or TOP expression. Depending on SQL Server's (or your server's) ability to focus the query on a single row and how many rows are included in your rowset, your pessimistic lock might block not only those accessing the row you're focused on, but the entire page, extent, or even table it resides in. That's the primary reason pessimistic locks are so dangerous—if not designed carefully, they can easily "cripple" an entire database.

## Using ADO.NET to create pessimistic locks

For those of you that haven't waded into ADO.NET, consider that the .NET data access interface is designed to create client-side, disconnected data structures and to leave no state on the server which might limit scalability or performance. ADO.NET expects to open a connection, return a rowset, and close the connection - leaving *no* state on the server. To use ADO.NET to construct a pessimistic lock, you're going to have to bypass that architecture and manually manage connection and transaction state—including your pessimistic locks.

Your ADO.NET connection "owns" your pessimistic lock(s)—when your connection is closed, the locks are released. However, when you close your connection in code, the .NET data provider connection pool merely returns the connection to the pool—the connection is not "reset" until another request is made for the same connection string by the same process. Not to worry, though. When you use the technique illustrated below, closing the connection releases any server-side locks as well.

## Coding Pessimistic Locks

To cook up a pessimistic lock you'll need to follow the following recipe. Once you've located a row to lock, you'll need to open a persistent connection, create a transaction object associated with the open connection and fetch and lock the row in question. By setting the transaction "IsolationLevel" option to *IsolationLevel.RepeatableRead*, ADO.NET and the provider knows that you want to lock this rowset from changes until closed—basically a pessimistic lock. Here's the code to accomplish this:

```
Sub LockRows()  
    Try  
        If cn2 Is Nothing Then  
            Else  
                If cn2.State = ConnectionState.Open _  
                    Then cn2.Close()  
                End If  
                cn2 = New SqlConnection(strConnect)  
                cn2.Open() 'This connection holds lock state  
                daLock = New SqlDataAdapter(strSQL, cn2)  
                'To construct UpdateCommand...  
                cb = New SqlCommandBuilder(daLock)  
                'To facilitate UpdateCommand  
                daLock.TableMappings.Add("Table", "Authors")  
                tran = cn2.BeginTransaction_  
                    (IsolationLevel.RepeatableRead)  
                daLock.SelectCommand.Transaction = tran  
                dsLocks.Clear() 'Clear any locked rows DataSet  
                daLock.Fill(dsLocks) 'Lock the specific row  
                Timer1.Enabled = True 'Start 20 second countdown  
                ShowLocks() 'Show impact on server-side SP_Locks  
                'Rows locked, cn2 holds conn. state, is left open  
                Catch ex As Exception  
                    MsgBox(ex.ToString)  
                End Try  
    End Sub
```

*Listing 1. Locking a specific row using a transaction.*

Note that the code creates a second connection (the first is used to display rows to change), creates a DataAdapter to fetch the row and provides a means to update it when the user decides to do so, creates a CommandBuilder to construct the action queries needed to change the locked row, and enables a timer to prevent the lock from remaining in place too long. When the timer counts down from "N" seconds, it rolls back the transaction and releases the pessimistic lock as shown below.

```
Private Sub Timer1_Tick(ByVal sender _  
    As System.Object, ByVal e As System.EventArgs) _  
    Handles Timer1.Tick  
    'Fires every 2 seconds  
    intCounter += 1
```

```

If intCounter > intTwoSecondWaits Then
    Timer1.Enabled = False
    intCounter = 0
    tran.Rollback() 'Rollback trans, release any locks
    ShowLocks() 'Display the status of any locks
    MsgBox("Locks were released...")
    lblTimerStatus.Text = ""
Else
    lblTimerStatus.Text = "You must commit the _
        change within " & _
        (20 - (intCounter * 2)).ToString & " seconds."
    End If
End Sub

```

*Listing 2. Releasing the pessimistic lock by rolling back the transaction.*

If the user chooses to commit the change (before 20 seconds), the following routine updates the data row(s), commits the transaction and in the process, releases the lock(s).

```

Private Sub UpdateCommitRow(ByVal sender _
As System.Object, ByVal e As System.EventArgs) _
Handles btnCommit.Click
    Try
        Timer1.Enabled = False
        lblTimerStatus.Text = ""
        CommitChangedRow()
        daLock.Update(ds) 'Post changes in DataGrid to db
        tran.Commit() 'Commit trans, release any locks
        dgRows.DataSource = ds.Tables(0) 'sho updated rows
        'Clear the "rows to lock" grid
        dgRowsToLock.DataSource = Nothing
        lblSQL.Text = ""
        ShowLocks() 'Display the status of any locks
        gbLock.Enabled = False
    Catch ex As Exception
        MsgBox(ex.ToString)
    End Try
End Sub

```

*Listing 3. Releasing the pessimistic lock by committing the transaction.*

Another interesting aspect of this application is its ability to display the locks as they're created. When the user clicks the "Lock" button, the code establishes the Transaction and pessimistic locks on the selected rowset. Depending on how many rows are involved, and how SQL Server decides to escalate the locks, the number of locks might span the entire table. By running the SQL Server stored procedure `sp_lock`, you can see which locks are in place before and after the Transaction and locks are created. The code to do this is as follows:

```

Sub ShowLocks()
    dal = New SqlDataAdapter("sp_lock", cn)
    dal.SelectCommand.CommandType = _
        CommandType.StoredProcedure
    dsLocks.Clear()
    dal.Fill(dsLocks)
    DataGrid1.DataSource = dsLocks.Tables(0)
End Sub

```

*Listing 4: Displaying server-side locks using sp\_lock.*

The application dumps the resultset from `sp_lock` as shown in Figure 1—before any locks have been established.

(sp\_lock before.gif)

*Figure 1. sp\_lock output before any transactions are started.*

When the user decides to lock the row (by clicking on the "Lock" button), the application runs the code (as shown above in Listing 1), and the pessimistic lock is created. When I run `sp_lock` again (see Figure 2), it's clear to see that there are several more locks established. Notice that our second connection's process (54) has several locks established—on a page, a key, on the database and on a table. The "IS" designation means that SQL Server wants to acquire a shared (S) lock or exclusive (X) lock on some of the object resources. For example, a shared intent lock placed at the table level means that a transaction intends to place one or more shared (S) locks on pages or rows within that table. Setting an intent lock at the table level prevents another transaction from subsequently acquiring an exclusive (X) lock on the table containing that page. See "Understanding Locking in SQL Server" in *SQL Server's Books Online* for more information on locking and how to decipher what `sp_lock` returns.

(sp\_lock after.gif)

*Figure 2. sp\_lock output before any transactions are started.*

### **Conclusion**

I hope that this brief overview of pessimistic locking helps those of you considering use of this feature enough information to do so wisely. Remember, pessimistic locks are not to be abused.