Locking Code

Peter Vogel

Last month, Peter Vogel examined the issues around locking records and concluded that, most of the time, you don't want to lock your records. In this article, he reviews that discussion and then shows the code that you can use to avoid record locking.

LAST month I went into tedious detail on when you should and shouldn't lock records. Specifically, I looked at when you needed to lock records early in the process. "Early in the process" means that records are locked either when they are first read or when the user first indicates that they intend to make a change to the record. Record locking early in the process, usually referred to as 'pessimistic locking,' should be avoided wherever possible, as it reduces your application's scalability—its ability to handle many users. As you'll see, writing code that avoids record locking also speed up your updates.

I claimed that record locking should only be done for changes when it's possible that:

- two users will update a record at the same time.
- a user will update a field that's related to a second field in the same record.
- the user won't update the second field in the same record.
- there's a business rule that would prevent the update to the first field based on data in the second field (in other words, some combinations of data in the two fields in the same record are forbidden); or
- you're updating every field in a record, even if it isn't changed.

For deletes, records only needed to be locked if there were one or more fields in the record that would prevent a record from being deleted (that is, there's a business rule that prevents a record in a specific state related to fields in the record from being deleted).

For multiple record updates, I encouraged you to use transactions to manage updating several records rather than using record locking with all its convoluted code. As you'll see, the code that I'll show you here is easily extended to handle multiple record updates through transactions.

In this article, I'm going to concentrate on the code necessary for implementing the SQL commands that will update only changed fields. See the sidebar "Access Object Locking" for what you can do in this area in data-bound Access forms.

Before looking at these SQL and code-based solutions, I should point out that if your back-end database supports triggers, then you should consider using those to avoid record locking. As I said before, locking is only required if there's a business rule specifying invalid field combinations (in an update) or deleting a record in a specific state. If your database supports triggers it might be a better idea to incorporate these business rules into server-side trigger code.

One final caveat: in the following code I've assumed that all the fields in my recordsets are string fields. For numeric and date fields you'll need to add code to handle placing the right delimiters around the data. I've just assumed that everything should be enclosed in single quotes.

Insert and deletes

I'll begin with the simplest cases: inserting a new record and deleting an existing record. Inserts can be ignored—there's no existing record to lock. Deletes, however, may require a record to be locked if it's possible that a field that the user doesn't update can prevent the record from being deleted.

The scenario that most programmers have in mind when locking for delete is this one:

1. User 1 retrieves the record and decides to delete it.
2. User 2 retrieves the same record and updates the DoNotDelete field.
3. User 2 saves the changes, gets no message, and is convinced that the record is protected.
4. User 1 deletes the record.

I don't consider it a problem if user 2 saves the changes after user 1 deletes the record. I have two reasons for this devil-may-care attitude:

- First, it's a fact of life that the person who saves their record first, wins. If user 1 saved on Monday and user 2 tried entering the data to prevent the update on Tuesday, we wouldn't feel that anything has gone wrong.
- More importantly, user 2 will get a message that the change wasn't made because the record had been deleted.

This scenario can be handled without record locking. The typical Delete statement in this scenario would

look like this:

```
Delete
    From Employees
    Where EmployeeId = 2001
```

In other words, the delete command would retrieve a record using the record's primary key and then delete it. By extending the Where clause, however, the problem in the scenario described above can be avoided:

```
Delete
    From Employees
    Where EmployeeId = "23149"
      And DoNotDelete = "False"
```

In this version of the Delete statement, the code checks at the time of the delete that the critical field hasn't been set. If user 2 has successfully changed the field, the Where clause will fail and the record won't be deleted.

As I noted, the failure to delete shouldn't necessarily be done quietly—under some circumstances, user 1 should be notified that the delete didn't succeed. There are two reasons that the delete could fail:

1. The DoNotDelete field was set.
2. The record no longer exists (that is, the record has been deleted or its primary key value changed).

If the command failed because the record was deleted, then user 1 doesn't need to be notified—their goal has been met. However, if the delete failed because of the DoNotDelete field being set, then the user should be told because the record is still in the database.

The code to handle this, using ADO, would look like the following. I use the already open AccessConnection object's Execute method to issue my delete command. The second parameter to that method must be a variable and it'll be updated with the number of records deleted. If that variable (ingRecs) is 0, then I have a problem and I need

Access Object Locking

With data-bound forms, you're constrained to the three settings in form and query's RecordLocks setting: No Locks, All Records, and Edited Record.

No Locks is optimistic locking with the record checked for changes upon update. If the record has been changed since the user retrieved it, at update time the message box shown in **Figure 1** will be displayed. Notice that the box gives the user the ability to write over the changed record. To my mind, this makes the warning almost useless, since I suspect many users will just click on the Save Record button. This is a problem because Access writes out all the fields displayed on the form, not just the ones that are changed.

The Edited Record setting locks the record when the user makes a change to any field. The record can no longer be updated from code, in a datasheet, or from another form once the user starts making changes. However, you have no protection against other applications changing the record after it's displayed but before your user starts making updates. To address this, if someone changes the record Access attempts to notify all interested parties by pushing out the new version of the data. If you suddenly notice, for instance, that data in a text box has changed and you don't remember doing it, it's probably because someone else has changed it and your form has picked up the new data.

The All Records setting locks every record in the table—the worst possible choice for scalability.

What's the right answer? What I've been doing is setting RecordLocks to No Locks to get optimistic locking. In the form's BeforeUpdate and Delete events I issue a minimal Select statement using the Where clause that I would have put in the equivalent SQL statement to check the DoNotDelete fields. For instance, in the Delete event, I would use:

```
Dim rec As ADODB.Recordset
Dim strSQL As String

Set rec = New ADODB.Recordset
strSQL = "Select 1 From Employees " & _
    "Where EmployeeId = '" & Me.EmployeeID & _
  "' And DoNotDelete = " & Me.DoNotDelete & ";"
rec.Open strSQL, _
     Application.CurrentProject.AccessConnection
If rec.EOF Then
    Cancel = True
    MsgBox "Record Not Deleted"
End If
```

If my Delete statement fails, I cancel the operation. I use similar code in those updates where I feel obliged to check related, unchanged fields. There are a couple of problems here, but the important one is that the recordset retrieved from a form that's based on an MDB file is a DAO recordset—the OriginalValue property isn't available.

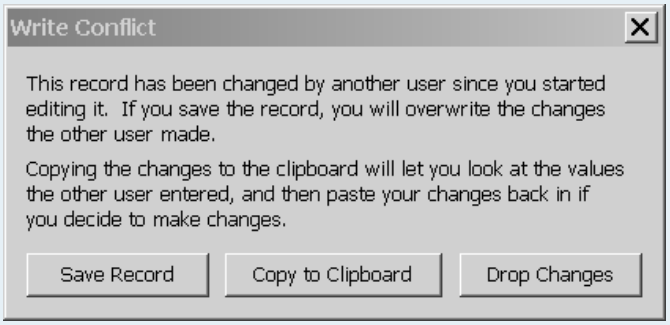

Figure 1. The Access Write conflict dialog box.

to check to see if the record has been deleted. If no record has been deleted, I will check to see if the record is still there and notify the user (remember, this part of the code is only executed in the rare instances where two users are updating the same record):

```
Dim ingRecs As Integer
Dim rec As ADODB.Recordset
Dim con As ADODB.Connection
Set con = Application.CurrentProject.AccessConnection

strSQL = "Delete From Employees " & _
  "Where EmployeeId = '" & rec("EmployeeId") & "'" & _
   " And DoNotDelete = '" & rec("DoNotDelete") & "'"
con.Execute strSQL, ingRecs
If ingRecs = 0 Then
  strSQL = "Select 1 From Employees " & _
     "Where EmployeeId = '" & rec("EmployeeId") & "';"
    Set rec = con.Execute(strSQL)
    If rec.EOF = True Then
           MsgBox "Record Not Deleted"
    End if
End If
```

It's possible that the DoNotDelete field could contain a wide variety of values that prevent the record from being deleted. If that's the case, it's probably not reasonable to check for all of those values in the Delete's Where clause. In this scenario, it's probably smarter to check that the DoNotDelete field hasn't been changed since user 1 retrieved the record (I'm assuming that code in the program would prevent user 1 from even trying to delete the record if it weren't in the appropriate state).

To build the Where clause in this case, you'll need to retrieve the value in the field when user 1 retrieved it. Simply going to the recordset and retrieving the current value isn't sufficient. After all, user 1 may have changed this value before deciding to delete the record. Fortunately, the OriginalValue property of the field in an ADO recordset will give you the value of the field when it was originally retrieved. The resulting code looks like this:

```
strSQL = "Delete From Employees " & _
  "Where EmployeeId = '" & rec("EmployeeId") & "'" & _
   " And DoNotDelete = '" & _
          rec("DoNotDelete").OriginalValue & "'"
```

To be able to use the OriginalValue, the recordset must not be a read-only recordset.

You can now delete without locking and without qualms.

I said at the start of this article that using this code would also speed up your updates. To begin with, if you don't lock your records as you retrieve them, your Select statements will run faster. However, you are obliged to check that the DoNotDelete field hasn't changed since you retrieved it before allowing the delete. Some programmers wait until the user starts to change the record. At that point, they re-fetch the record to lock it and retrieve the current value of the DoNotDelete field. This incurs another trip to the database server—probably the slowest thing that your application can do. The method that I'm suggesting here does the check as part of the update, eliminating that extra trip.

One case does remain to be discussed: what if user 1 explicitly sets the DoNotDelete field to a value that permits deletes while, at the same time, user 2 is setting the field to a value that doesn't permit deletion? Should user 1's change be allowed, overriding user 2's change? My feeling is that there's no way to arbitrate this dispute programmatically. My answer is that user 1's scenario shouldn't be allowed—the user should be obliged to save the record with the new value in the DoNotDelete field before deleting the record is enabled.

Which leads to the next section in this article, handling updates without locking.

Updates

With updates, the first step in avoiding locking is to create a SQL statement that updates only the changed fields. As with the Delete query, the trick here is to add clauses to the SQL statement only where the original version of the data differs from the current version. This code goes through each field in the record, building the update portion of the SQL statement based on the changed fields:

```
Dim fld As ADODB.Field

For each fld in rec.Fields
    If fld.Value <> fld.OriginalValue Then
      strSQL = strSQL & fld.Name & " = '" & _
                fld.Value & "', "
  End If
Next
If strSQL > "" Then
  strSQL = Left(strSQL, Len(strSQL)-2)
  strSQL = "Update Employees Set " & strSQL & _
    " Where EmployeeId = '" & rec("EmployeeId") & "';"
   con.Execute strSQL, ingRecs
End If
```

As with the Delete statement, you can check to see if other, related fields have changed before doing your update by using those fields with their original values in a Where clause. In my example from last month, I discussed changing an employee's salutation (Mrs., Mr., Dr.) and last name. These two fields are related, but it's possible that you might change one without the other. For instance, upon marrying, a woman might change both her salutation and her last name. Alternatively, upon acquiring her doctorate, she might change only her salutation. As a result, in my code, if the user changes Salutation field but not the last name, I may want to extend my Where clause to make sure that salutation field hasn't been updated. As with the Delete, if an Update statement fails you'll want to issue a Select statement to see if the record has been deleted to provide the right message to the user.

Extending this code to handle multiple record updates isn't hard to do. The key, as I noted last issue, is

to begin a transaction before the first record is updated. In the code that I've shown up until now, if the update doesn't complete, the user is notified. In multiple record updates, you just need to add a transaction rollback to that notification to back out your changes:

```
con.BeginTrans
…update code…
If ingRecs = 0 Then
    …notification code…
    con.RollbackTrans
Else
    con.CommitTrans
End If
```

Writing to fail

You may have noticed that I've written my update queries so that they will fail. In my deletion example, if the DoNotDelete field has been changed then my SQL Delete statement won't succeed. When I'm outlining these anti-locking strategies, I'm sometimes asked if it wouldn't be better to ensure that the command will succeed before executing it. It's a good question.

However, any other solution than the ones that I've shown here will involve either pessimistic locking or an extra data retrieval. My answer is that, in most applications, the typical situation is that only one user will be using a record at any time. The vast majority of the time, therefore, the Delete will succeed. It seems foolish to me to add extra overhead to my system to ensure success for the few occasions where two users are fighting over the same record. I would prefer to treat those situations as they are—as special and infrequent cases.

I don't want to suggest that there is no place for locking. If, for instance, your typical case is that multiple users are handling any record, then locking records as they're retrieved would probably make sense. With the techniques that I've shown here, there's a possibility that a user will expend a lot of time and effort in changing a record only to have that work thrown away when I'm unable to save their data. Guaranteeing updates with locking may be a better strategy if that's a frequent occurrence. It may also be that there's a scenario that I haven't considered here that requires you to use pessimistic locking.

However, locking records should always be your last choice if you want to build highly scalable systems. And even if you haven't considered pessimistic locking, the techniques that I've shown here will improve your data integrity. ▲

DOWNLOAD **LOCKING.ZIP at www.smartaccessnewsletter.com**

Peter Vogel (MBA, MCSD) is the editor of *Smart Access* and is a principal in PH&V Information Services. PH&V specializes in system design and development for systems that use Microsoft technologies. Peter has designed, built, and installed intranet and component-based systems for Bayer AG, Exxon, Christie Digital, and the Canadian Imperial Bank of Commerce. He also wrote *The Visual Basic Object and Component Handbook* (Prentice Hall, currently being revised for .NET). In addition to teaching for Learning Tree International, Peter wrote its Web application development, ASP.NET, and technical writing courses, along with being technical editor of its COM+ course. His articles have appeared in every major magazine devoted to VB-based development, can be found in the Microsoft Developer Network libraries, and will be included in Visual Studio .NET. Peter also presents at conferences around the world. peter.vogel@phvis.com.