vb123.com

SmartAccess

Solutions for Microsoft® Access® Developers

Navigating Through Recursion, Part 1

Christopher R. Weber

Knowing your way around your Access project is important for any developer. In this first of two installments, Christopher Weber takes us through a navigation map generating algorithm he uses to populate a table that describes how the forms and reports in an Access database relate to each other. In next month's issue, Chris will demonstrate recursive query and reporting techniques he uses to generate a tree navigation map of the database.

HAVE you ever taken over a large Access project and been overwhelmed in the first requirements meeting by the plethora of synonyms used to talk about a Client / Customer / Whatever entity? Then there's the corresponding jargon for the Client screen, the Customer form, or the Whatever interface. And everyone in the meeting expects you to know how to navigate through the product you've barely seen.

I've walked into this situation at least three times in the past year: once taking over a project started by another developer, once consulting on a project managed by another developer, and once being added to a team of developers on a very large Access project. Invariably, the project is thrashing to catch up to an arbitrary schedule, there's little or no documentation, and the work needs to be finished yesterday. It's always the same problem ("the Client screen needs to have such and such") and I'm always asking the customer, "Okay, show me how you got there." At some point, I have to sit down and map out all of the navigation routes possible in the project, either on paper or in Visio.

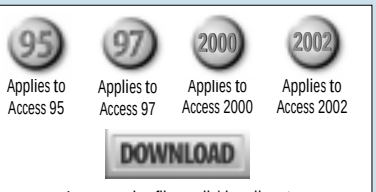
Wouldn't it be nice to have a form like the one in **Figure 1** (on page 4) that shows you all the links between interfaces (forms and reports) in the database? Wouldn't it be nice to have the data generated for you automatically?

Mapping your interfaces isn't a trivial matter and really should be done in the architectural stage of requirements gathering in order to limit the product's

Continues on page 4

January 2003

Volume 11, Number 1



Accompanying files available online at www.smartaccessnewsletter.com

In code, an underscore (_) as the last character of a line indicates that the line has been wrapped for layout purposes. In Access 95 and up you can use the code as it appears, but in Access 2.0 you must recombine the wrapped lines.

So Here's My Plan

Peter Vogel

LIFE keeps getting more complicated for the editor of a Microsoft newsletter. To begin with, there's my obligation to meet the needs of a diverse group of Access developers (intermediate to advanced, lone developers, members of a programming team, consultants, and corporate developers). Then there's the issue of supporting the two to four different versions of Access (95/97, 2000/2002)—with a new version on the way. While many developers are very happy with Jet/MDB development, others have moved on to client/server development using SQL Server/MSDE or other database management systems (say, Oracle). On top of that there's the variety of packages that Access developers may want to integrate with: Outlook, Microsoft Office, Visual Basic, .NET Framework, among other third-party packages. Finally, of course, there's always the Web (I shouldn't say finally—there's probably lots of other things that you want to know about that I haven't listed here). It's tough.

I know—whine, whine, whine. If the job is such a problem, move on. But I like the challenge. As you've probably noticed, *Smart Access* is trying to meet all of those needs.

For instance, in the long run, there's probably a client/server database in your future. So we've been running articles aimed at Jet/MDB developers who want to learn what they'll need to build client/server applications within the Access environment. Russell Sinclair has been helming that effort most recently. For instance, one of the major limitations of Access Data Projects is the absence of tools to manage SQL Server. Last month Russell filled that gap (at least as far as managing security) with an Access add-in.

For a number of years we've helped you with understanding SQL with the "Working SQL" column. This month, Russell starts a new series to give Access developers the same depth of understanding with SQL Server's programming language native to SQL Server: T-SQL. Russell approaches this topic from the point of view of an Access developer rather than a SQL Server developer.

For most of the past year Danny Lesandrini has been running our reviewer's corner, introducing you to new products that will make you more productive as a developer. For those of us who thought that the world of Access tools consisted of FMS's excellent product line and Speed Ferret, Danny has uncovered a wealth of tools to meet a variety of needs. For those who are considering client/server development and find the MSDE inadequate, Danny even got *Smart Access* readers a special deal to upgrade to SQL Anywhere.

However, the core of *Smart Access* remains articles on techniques for building applications with Microsoft Access. A great *Smart Access* article will show you how to use some part of the rich store of Access technology or how to solve thorny problems using Access. I'm always on the lookout for techniques that apply to every version of Access, but we've also had articles that showed you how to take advantage of features in specific versions of Access.

But there's a great big world out there, outside of Access. I think that the next version of SharePoint is going to be an important part of the Microsoft world, and that Access developers will find it as useful as Outlook. So Nikander and Margriet Bruggeman introduced you to it in *Smart Access*. We've also shown you how to use Web Services from Access and kept an eye on Access competitors like StarOffice. Coming up we have an article on using Access with Microsoft Terminal Services. Our goal is not only to help you program better, but to make sure that you know what's in your environment. You may not need all the detail in these articles, but you'll understand the issues and—when the time comes—the detail will be waiting for you to use (you do keep your back issues, don't you?).

My plan for covering the next version of Access is the same plan that we used with Access 2002. *Smart Access* will feature a series of articles (no more than one per issue) that will discuss, one by one, the new features in Access. You'll understand how the feature works (if it does), what it does for you, and whether it's valuable enough for you to consider upgrading. However, the core of *Smart Access* will remain with the versions of Access that you're using right now: 95/97, 2000/2002.

Which brings me to the point: How are we doing? Is this a useful mix for you? What do you need to know? These are tough economic times and my goal is to make *Smart Access* your best friend when working with Access. Let me know what you need: E-mail me at peter.vogel@phvis.com. I'm looking forward to hearing from you. ▲

Recursion, Part 1...

Continued from page 1

complexity. A couple of years back, I watched a company that was developing sophisticated banking software go under because no one could predict all the paths the user could take through their product. The owner / domain expert wanted maximum flexibility for the user, and designed the product to jump from almost any interface to any other. The need was clear in his mind, but none of the users could fathom it. The development team, who weren't bankers, was constantly redesigning the application because of recursive calls from one interface to the next. A called B, which called C, which reopened A again, and so on. Though it was the reality in the banking world that these entities were interrelated in a hyper manner, building software to mimic that reality became a cumbersome chore that eventually collapsed on itself. Worst of all, much of this recursion wasn't "discovered" until beta testing. Clearly, this was a classic failure caused by ignoring software engineering principles.

Knowing where the product begins and ends along with everything possible in between is certainly one step toward successful completion of a project. However, there are times when that map isn't known. For example, when an evolutionary prototyping approach is being taken, the design may grow as the project continues. The banking project was using this approach, which certainly got the project going at the beginning, but eventually led it down an evolutionary dead end. At some point, the project should have abandoned that model and taken on a more structured approach. And, as I stated earlier, the map may not exist when you have to take on an existing project.

The Access advantage

Because an Access project typically stores all of its forms and reports in a single .mdb file, and because Access exposes all of the objects and their properties to the developer, the information for tracing each route through a database exists in the .mdb file and can be gleaned to create a map of the system. There are, however, some presumptions in this approach that I must be clear about:

- Navigation is limited to moving between forms and reports (as well it should be). Users aren't directly exposed to queries and tables except as subforms / subreports (in 2000 onward).
- You know the name of the opening interface(s), whether it's a built-in Access switchboard or a custom form.
- Navigation is assumed to be primarily a branching mechanism, organized hierarchically: The user enters at a specific point and then chooses to move to one of a list of forms. Hyper-navigation (where any form can lead to any other form) between interfaces is limited. Any map of a hyper-navigation scheme is arbitrary as to where navigation begins and ends. Though the navigation path could be mapped in multiple views using my technique, they aren't the focus of this discussion.
- Navigation links are created in code using the DoCmd object's OpenForm and OpenReport methods, through custom functions that receive the name of the object to open, and through hyperlinks. No macros are used and calls to forms and reports in public modules (a rare case) aren't accommodated.
- The system may use DoCmd or custom functions to open objects whose names are exposed in a list or some control property. For example, the user can select one or more reports to print from a list box of possibilities, or click on a check box whose Tag property contains the name of the report to preview. You, as developer, must be capable of filling in the blanks where my algorithm cannot trace such designs.
- You don't want to repeat branches of the map that have already been exposed at some lower level, resulting in repetitive or recursive mapping of a branch. However, there are times when this is unavoidable.
- I'm presuming the use of DAO for object coding, and haven't used any Data Access Pages. For purely Access applications, this is typical and most efficient.

This list may seem somewhat limiting, but I've found that it covers the vast majority of navigation routes in the databases I've tested. Indeed, this may be a byproduct of my development style, but I presume that my techniques are typical of most Access developers.

Figure 1. zsfrmNavigationMap showing unconfirmed entries found through dynamically loaded subforms and parameterized OpenForm methods.

Where do you begin?

The first thing that you need is a way to store navigation link information. The answer is, of course, an Access table with a simple design, as shown in **Figure 2**.

The fields in zstblNavigationMap describe a self-joining hierarchical relationship between two objects: the CallingObj and ObjCalled. For instance, a switchboard (listed in CallingObj) could call a form named frmOrders (which would be listed in ObjCalled). The tables also describe the branching level at which the calling object is found. In my example, the switchboard would be at level 1 and links from frmOrders to other forms or reports would be at level 2, and so on. The Confirmed Boolean field in the table states whether the link found was verified (in this example, whether the ObjCalled was found to be an existing form or report in the database). SystemGenerated is set to true if the algorithm found the linkage. The field is left false for entries the user might have to make manually.

The table has several indexes (see **Figure 3**). The primary key, an autonumber, will increment with each entered record and is useful for examining the order in which the algorithm found the links. A unique index on Level-CallingObj-ObjCalled ensures that the same entry isn't made more than once, and that combinations of CallingObj and ObjCalled found at earlier levels can be used to suppress mapping of the same links found at higher levels. CallingObj and ObjCalled are each indexed because I'll be joining these fields to each other in a self-join query later on.

Lastly, to avoid aberrant entries, the table has a validation rule set in its property sheet that the CallingObj can't be the same as the ObjCalled. This shouldn't happen in the mapping algorithm, but I've seen it pop up. What's more, users shouldn't make such entries (as shown in **Figure 4**).

Figure 2. zstblNavigationMap in design view.

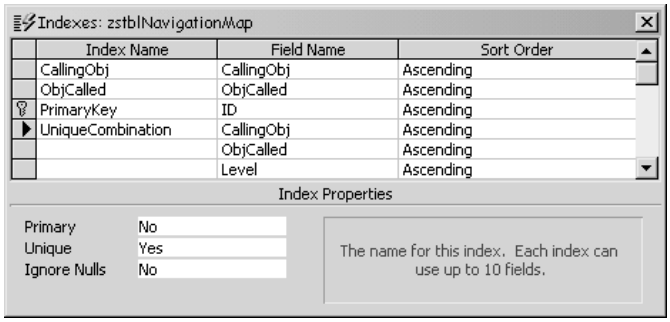


Figure 3. zstblNavigationMap Indexes window.

Filling the mapping table

To fill the table with mapping records, you should, theoretically, be able to recurse through the linkages between objects and enter records for each link found. Then, by following the records in Level-CallingObj-ObjCalled order, you can record how each object relates to the other objects. This turns out to be easier said than done.

To begin the process, I'll look at a procedure that traces a Switchboard table generated by the built-in Switchboard Manager menu option under Tools | Database Utilities. The routine cmdLoadSwitchboardTableEntries_Click loops through records in the Switchboard table that have a Command value of 2, 3, or 4 (Open form in Add mode, Open form in Edit mode, or Open report in preview). For each entry, cmdLoadSwitchboardTableEntries_Click calls my routine LogItAndDrillIn with the name of the ObjCalled (in this case, the Argument field in table Switchboard enclosed in double quotes), a pointer to a recordset for editing the table zstblNavigationMap, a Level value of 1, the type of object found (acForm or acReport), and the name of the CallingObj (in this case, "Switchboard"):

```
Private Sub cmdLoadSwitchboardTableEntries_Click()
On Error GoTo ErrorHandler
Dim rst As Recordset
Dim rstSwitchboard As Recordset
Dim intObjectTypeFound As Integer

If genObjectExists("Switchboard Items", acTable) Then
  Set rstSwitchboard = CurrentDb.OpenRecordset( _
    "SELECT Command, Argument FROM " & _
    "[Switchboard Items] WHERE Command in(2,3,4)")
    Set rst = CurrentDb.OpenRecordset( _
                "zstblNavigationMap", dbOpenDynaset)
  With rstSwitchboard
    Do While Not .EOF
      Select Case !Command
        Case 2, 3
          intObjectTypeFound = acForm
        Case 4
          intObjectTypeFound = acReport
      End Select
      Call LogItAndDrillIn("""" & !Argument & _
          """", rst, 1, intObjectTypeFound, "Switchboard")
      .MoveNext
    Loop
  End With
Else
  MsgBox "Switchboard Items table not found.", _
          vbCritical + vbOKOnly, "Process halted..."
End If
subNavigationMap.Form.Requery

Exit_Here:
```

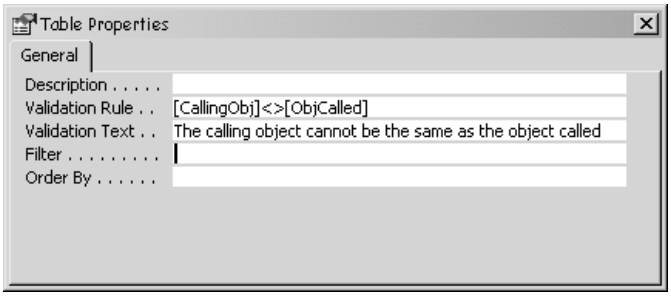


Figure 4. zstblNavigationMap Properties window.

```
  If Not rst Is Nothing Then Set rst = Nothing
  If Not rstSwitchboard Is Nothing Then
        Set rstSwitchboard = Nothing
  End If
Exit Sub

ErrorHandler:
  MsgBox "Error #" & Err.Number
  Resume Exit_Here

End Sub
```

LogItAndDrillIn() attempts to open the ObjCalled and examine it for further linkages. The reason the name of the object being called is passed in using double quotes is because the name will sometimes be embedded in a DoCmd.Open*object* line of code and be found in the first set of double quotes found in the line. I extracted the name in the first few lines of code. If a pair of double quotes can't be found, the reference is trimmed and marked for later manual discovery. An example might look like the following, where list box lstReports allows the user to open one of several available reports kept in a value list or a system table:

```
DoCmd.OpenReport lstReports, View:=acPreview
```

The routine to handle this code looks like this:

```
intFirstQuotePosition = InStr(strReference, Chr(34))
If intFirstQuotePosition Then
  intSecondQuotePosition = _
      InStr(intFirstQuotePosition + 1, strReference, _
      Chr(34))
  strNameOfObjectFound = Mid(strReference, _
    intFirstQuotePosition + 1, intSecondQuotePosition _
    - intFirstQuotePosition - 1)
Else
  strNameOfObjectFound = Trim(strReference)
End If
```

In the next step, LogItAndDrillIn() checks for existing entries in zstblNavigationMap with either the same CallingObj or ObjCalled at a lower level, because you don't want to log an object at a level higher than one already found. If you do, you may see entire sets of branches in the map repeat at various levels and, possibly, recursively call themselves—resulting in an endless hierarchy. Remember that banking software? A hyper structure would cause the same result. However, it may be too late to prevent this from happening by the time the new route is found.

How so? In my example, my Switchboard opened frmOrders. Now frmOrders could have a link to open frmCustomers. The frmOrders–frmCustomers link would be logged at level 2. If Switchboard also has a direct link to frmCustomer, a record with frmCustomers as the ObjCalled will be added for frmCustomers at level 1 (a link from Switchboard) in addition to the link to frmCustomers at level 2 (previously found called from frmOrders). My code prevents that second entry from being made.

This is a design tradeoff that I accepted because I always want to see the earliest call tree for an object, but not necessarily for any later repetitive branchings of the same object. This doesn't prevent seeing frmCustomers called at later levels, because the software checks for the current ObjCalled as the CallingObj at earlier levels. It merely stops the continuation of duplicate branchings of the navigation tree:

```
rst.FindFirst "(CallingObj='" & strNameOfObjectFound _
    & "') AND (Level<" & intFromLevel & ")"
fNotFoundPreviously = rst.NoMatch
If fNotFoundPreviously Then
  rst.FindFirst "(CallingObj='" & _
    pstrNameOfCallingObj & "') AND _
    (Level<" & intFromLevel & ")"
  fNotFoundPreviously = rst.NoMatch
End If
```

Before I log any entries, I check to be sure the ObjCalled exists. This is handled by the genObjectExists routine in the class module, a very handy piece of generic code that you may find useful. I then add a record to zstblNavigationMap through the rst pointer, indicating whether the object can be found in the database and that this is a system-generated entry:

```
If fNotFoundPreviously Then
  fObjExists = genObjectExists(strNameOfObjectFound, _
                    intObjectTypeFound)
  rst.AddNew
  rst!Level = intFromLevel
  rst!CallingObj = pstrNameOfCallingObj
  rst!ObjCalled = strNameOfObjectFound
  rst!Confirmed = fObjExists
  rst!SystemGenerated = True
  rst.Update
```

Now the recursion begins. First of all, remember that I passed in the recordset pointer to LogItAndDrillIn. I pass the reference in because I don't want to create multiple, unresolved pointers to zstblNavigationMap, which would be resource-intensive. It could also, possibly, leave me with records hanging in the midst of an edit should I somehow choose that route while programming. Secondly, I don't continue searching along the path exposed by ObjCalled unless it was passed in with quotes around it *and* it was found to exist in the earlier steps. In other words, I don't want to recursively search on a line like DoCmd.OpenReport lstReports, View:=acPreview where the actual name of the object(s) being called must be discovered manually.

Up to this point, I've been discussing entries in a table used by the switchboard program. The continuation of the search is handled by the routine NavigationMap, which receives the Level + 1, the name of the object found, the same name as the object to examine, and the type of object (acForm or acReport):

```
If intFirstQuotePosition And fObjExists Then
   NavigationMap intFromLevel + 1, _
       strNameOfObjectFound, strNameOfObjectFound, _
       intObjectTypeFound
End If
```

One last thing to discuss before you leave LogItAndDrillIn(): the error handler for error number

3022 (duplicate entry in a unique index). Should another branch call ObjCalled at the same level from the same CallingObj, the unique index would be violated. The unique index stops the entry, so you can just ignore the error and exit the routine:

```
Exit_Here:
  Exit Sub

ErrorHandler:
  Select Case Err
    Case 3022
      'duplicate entry in unique index
    Case Else
      MsgBox "Error #" & Err.Number
  End Select
  Resume Exit_Here

End Sub
```

Navigating through recursion

NavigationMap begins its processing by first verifying that both the CallingObj and ObjCalled exist in the database. This isn't redundant, as manual entries can be traced by NavigationMap(). In fact, if the database doesn't use table Switchboard, or has an alternate entry point, a call can be made through the interface to begin the tracing process:

```
If Not (genObjectExists(pstrNameOfCallingObj, _
          acForm) Or _
   genObjectExists(pstrNameOfCallingObj, acReport)) _
   Then
    MsgBox "Object " & pstrNameOfCallingObj & _
      " NOT found.", vbCritical + vbOKOnly, _
      "Process halted..."
    Exit Sub
End If

If Not genObjectExists(pstrNameOfObjectToExamine, _
                       pintObjectType) Then
  Select Case pintObjectType
    Case acForm
      MsgBox "Form " & pstrNameOfObjectToExamine & _
       " NOT found.", vbCritical + vbOKOnly, _
       "Process halted..."
    Case acReport
      MsgBox "Report " & pstrNameOfObjectToExamine & _
       " NOT found.", vbCritical + vbOKOnly, _
       "Process halted..."
    End Select
  Exit Sub
End If
```

If found, a call to GetObjectToExamine opens the object for examination and then minimizes it:

```
Set rst = CurrentDb.OpenRecordset( _
   "zstblNavigationMap", dbOpenDynaset)
GetObjectToExamine pintObjectType, _
   pstrNameOfObjectToExamine, objFrmRpt
```

If the object has a class module, each non-comment line of code is examined for OpenForm and OpenReport methods. The code also checks for a pair of custom functions, OpenAForm and OpenAReport, that I use to handle error 2051 that occurs when Cancel is set to true in a form's OnOpen event or a Report's NoData event. Both examples can be found in the database in this article's accompanying Download file (available at www.smartaccessnewsletter.com). Leaving the lines that handle this in the NavigationMap() routine won't hurt anything. However, if you have similar implementations that receive the name of the object to open, simply replace the references found here with your own. In either case, if found, the line of code is passed to LogItAndDrillIn where it's parsed for the object name in double quotes, along with the other required parameters:

```
If objFrmRpt.HasModule Then
  Set mdl = objFrmRpt.Module
  With mdl
    For i = 1 To .CountOfLines
      intObjectTypeFound = 0
      If Not Left(Trim(.Lines(i, 1)), 1) = "'" Then
        If InStr(.Lines(i, 1), "OpenForm") Or _
             InStr(.Lines(i, 1), "OpenAForm") Then
          intObjectTypeFound = acForm
        ElseIf InStr(.Lines(i, 1), "OpenReport") Or _
            InStr(.Lines(i, 1), "OpenAReport") Then
          intObjectTypeFound = acReport
        End If
        If intObjectTypeFound Then
          LogItAndDrillIn(.Lines(i, 1), rst, _
            pintFromLevel, intObjectTypeFound, _
            pstrNameOfCallingObj)
        End If
      End If
    Next i
  End With
End If
```

After examining the object's module, each control on the object itself is examined for calls in its event properties. But first, another call to GetObjectToExamine is made to reopen any objects that were inadvertently closed during the recursion process. This shouldn't be possible, but I've seen the DoCmd.Close action at the end of this procedure close objects that were opened several levels earlier in the recursion and were supposedly out of scope. This failsafe code ensures that you continue examining the current object of interest.

Any findings are passed to LogItAndDrillIn(), which continues the process recursively:

```
GetObjectToExamine pintObjectType, _
      pstrNameOfObjectToExamine, objFrmRpt
For Each ctl In objFrmRpt.Controls
  For Each prp In ctl.Properties
    intObjectTypeFound = 0
    If Left(prp.Name, 2) = "On" Then
      If InStr(prp.Value, "=OpenAForm") Then
         intObjectTypeFound = acForm
      ElseIf InStr(prp.Value, "=OpenAReport") Then
        intObjectTypeFound = acReport
      End If
    End If
    If intObjectTypeFound Then
      LogItAndDrillIn prp.Value, rst, pintFromLevel, _
           intObjectTypeFound, pstrNameOfCallingObj
    End If
```

Next, I check for hyperlinked objects whose reference can be found in a control's HyperlinkSubaddress property:

```
If prp.Name = "HyperlinkSubaddress" Then
  If Left(prp.Value, 4) = "Form" Then
    LogItAndDrillIn """ & Right(prp, Len(prp) - 5) _
      & """, rst, pintFromLevel, acForm, _
      pstrNameOfCallingObj
  ElseIf Left(prp, 6) = "Report" Then
```

```
    LogItAndDrillIn(""" & Right(prp, Len(prp) - 7) _
      & """, rst, pintFromLevel, acReport, _
      pstrNameOfCallingObj)
  End If
End If
```

Finally, I handle subforms and reports. However, as of Access 2000, you can't open a subform or subreport while its parent object has it open. Therefore, I must first determine whether the subform's SourceObject exists. If so, I remove the reference in the subform control so that the name of the object can be passed recursively to NavigationMap. I don't, however, call LogItAndDrillIn to record the link from the form to the subform because I don't consider a subform or subreport as a new node on the navigation tree. For instance, if you ask a user what interface they're using, they won't cite the subform, just the main form. So too, neither will I:

```
If ctl.ControlType = acSubform Then
  If Len(ctl.SourceObject) Then
    If intObjectTypeFound = acReport Then
      If genObjectExists(Right(ctl.SourceObject, _
          Len(ctl.SourceObject) - 7), acReport) Then
        strObjectToExamine = Right(ctl.SourceObject, _
                      Len(ctl.SourceObject) - 7)
        ctl.SourceObject = ""
        NavigationMap pintFromLevel, _
          pstrNameOfCallingObj, strObjectToExamine, _
          acReport
      End If
    ElseIf genObjectExists(ctl.SourceObject, acForm) _
                Then
        strObjectToExamine = ctl.SourceObject
        ctl.SourceObject = ""
        NavigationMap pintFromLevel, _
           pstrNameOfCallingObj, strObjectToExamine, _
           acForm
      Else
        'not form or report, could be query/table
      End If
    Else
     'empty subform--set at runtime
      LogItAndDrillIn ctl.Name, rst, pintFromLevel, _
            intObjectTypeFound, pstrNameOfCallingObj
    End If
End If
```

I don't trace into source objects that are tables or queries as is now allowed in Access 2000 and forward. This is because neither have events that will lead to new nodes on the navigation tree. Lastly, I have the cleanup code and error handling:

```
Exit_Here:
On Error Resume Next
  If Not rst Is Nothing Then Set rst = Nothing
  DoCmd.Close pintObjectType, objFrmRpt.Name, acSaveNo
  Exit Sub

ErrorHandler:
  Select Case Err
    Case 7784, 2467
      'subform / subreport source object already open
      'object inadvertently closed in previous call
      Call LogItAndDrillIn "UNABLE TO OPEN " & _
          pstrNameOfObjectToExamine, rst, _
          pintFromLevel, intObjectTypeFound, _
          pstrNameOfCallingObj)
      Err.Clear
      Resume Exit_Here
    Case Else
      MsgBox "Error #" & Err.Number & ": " & _
          Err.Description & " by " & Err.Source, _
          vbOKOnly, "Error in procedure NavigationMap"
      Resume Exit_Here
  End Select
End Sub
```

Once the examination is finished, the object is closed. All of this opening and closing of forms and reports is visible on the screen, which I find both interesting and reassuring (it lets me know that something is going on). I also specifically handle error 7784, which arises when the subform or subreport I wanted to examine is already opened through examination of an object higher up the navigation tree. This is an obscure, but real threat to the process, and I've run into it on a test of a banking application (imagine that!).

Making it easy(er)

The form's zsfrmNavigationMap (you saw it back in Figure 1) and zsfsubNavigationMap expose all the mapping and maintenance methods for my data.

The subform zsfsubNavigationMap displays the records in zstblNavigationMap. The records appear, by default, in primary key order, which is the order in which they're discovered by the algorithm. I found this order useful for debugging and have left it that way. A click on the raised labels allows you to sort by the corresponding data fields using my ExplorerSort function included in a public module in the sample database. You can also use arrow keys to move up and down between records like a datasheet. This is enabled through the use of Form_KeyDown_AsDataSheet routines, found in the subform's module. All fields are editable except SystemGenerated. This allows you to make manual entries, corrections, and edits when needed.

The main form, zsfrmNavigationMap, contains the coded methods for extracting the map and maintaining zstblNavigationMap. The Clear Map button allows you to clear the map table of all records that are system-generated. Should you decide to clear the table and run the algorithm again, non-system-generated entries are *not* deleted, allowing you to avoid repetitive manual rediscovery of those entries the system might not find.

The Load Switchboard button calls cmdLoadSwitchboardTableEntries_Click(), the first procedure you examined. Delete Current Row deletes the current row in the subform. The Load Current Row button copies the entries in the current row of the subform into the text boxes at the top of the main form. Map It passes the parameters in the text boxes to NavigationMap(). So, should you have a custom switchboard or other opening form, you'd begin your examination of the database by entering 1, *formName*, *formName*, and selecting Form as the type of object called. Click Map It and you're on your way.

Continues on page 19

Understanding Triggers

Russell Sinclair

2000 2002

In the inaugural article in a new ongoing series, Russell Sinclair takes you through the concepts you need to know in order to understand what triggers are, what they do, and how to use them correctly in Access applications.

IF you ask an experienced SQL Server developer why you should use SQL Server as the back-end data store for a database application, triggers would likely come in high on their list of reasons. On my list, they would come first. But what are triggers and how are they useful to the Access developer?

Triggers are blocks of code that can be attached to some event occurring on data within a table or a view. They allow you to react to data being updated, deleted, or inserted into a table, regardless of whether that change is made from a form, a datasheet, a query, or VBA code. Based on the changes that have taken place, you can code triggers to react to or change the behavior of the update that was originally requested.

Let me be clear about what you'll learn in this article: I'm going to show you the why, when, and how of triggers. This is the "conceptual" trigger article that will give you the information that you need to decide whether you want or need triggers.

Anatomy of a trigger

Before you can use triggers, you need to understand what they do and when. Triggers react to data modification requests (INSERT, UPDATE, or DELETE) on tables and views. There are two major types of triggers in SQL Server 2000: AFTER (or FOR) triggers, and INSTEAD OF triggers. As you might guess, AFTER triggers are fired after the calling action has taken place, and INSTEAD OF triggers are fired instead of the calling action. If you're using SQL Server 7 or earlier, INSTEAD OF triggers aren't available and AFTER triggers are referred to as FOR triggers.

When an AFTER trigger is fired, the data modification requested by the user has already taken place. SQL Server will only fire an AFTER trigger if the data to be inserted has successfully passed other tests along the way. If the action violates referential integrity (RI) or check constraints, for instance, your trigger won't fire. So when you're coding AFTER triggers, keep in mind that you need to code your trigger as if the data modification has already happened. AFTER triggers can be defined on tables, but they cannot be defined on views.

INSTEAD OF triggers are fired instead of the requested action, so no data has changed at the time that they're fired. In fact, unless you specifically code a trigger to carry out the requested action (or some other action), no data modification ever takes place. The data modification that raised the trigger isn't validated against any referential integrity or check constraints (after all, it hasn't happened yet). This means that you may well have to perform your own validation to make sure that the data conforms to the rules defined in the database. If the code in an INSTEAD OF trigger modifies data in the original target table or view, the trigger won't be fired again. The data modification programmed into the trigger is allowed to go through, and then constraints will be checked by the table and any AFTER triggers will be fired.

When working with triggers, there are two tables with which you'll need to work: "inserted" and "deleted". These tables are exact mirrors of the table or view on which the trigger is defined. They act as temporary holding areas so that you can determine what data has changed. When an INSERT statement adds data to a table, the "inserted" table is filled with the records that have been (or will be) added. When a DELETE statement deletes data from a table, the "deleted" table is filled with the records that have been (or will be) deleted. In triggers, SQL Server treats UPDATE statements as a DELETE followed by an INSERT. This means that there will be data in both the deleted and inserted tables that reflects the changes requested to the data.

What the heck just happened?

One of the challenges you'll face when writing triggers is trying to figure out what data modification just took place. This is because you can define triggers that fire for one, two, or all three of the data modification statements (UPDATE, DELETE, and INSERT). Triggers are fired once for each data modification statement that takes place, regardless of the number of records that statement affects. For example, this statement deletes all of the Orders for one Customer with a single statement with the code:

```
DELETE FROM Orders WHERE CustomerID = 'ALFKI'
```

This could delete one or more rows, but the DELETE trigger will only fire once. If this statement deletes 10 records from the Orders table, the deleted table will contain all 10 of the affected records and your

trigger will fire only once. This means that you must code any reaction to this event so that it handles all 10 records properly.

Determining the action that raised a trigger is easy if the trigger is only defined against one action. If the trigger is defined to handle more than one type of statement, determining the modification that took place can be tricky. Fortunately, two SQL Server functions and some creative joins can make this process easier.

When you code triggers that respond to UPDATE statements, you often want the trigger to perform some action only if a particular column or columns were modified. This can be determined using the IF UPDATE clause. This clause takes a column name as a parameter and checks whether the specified column was modified. In order to determine if multiple columns were updated, you can use multiple UPDATE(ColumnName) statements together to determine whether the columns were updated. For example, the code that follows this IF statement will execute only if the CustomerID and the OrderTotal columns were modified by the statement that fired the trigger:

```
IF UPDATE(CustomerID) AND UPDATE(OrderTotal)
```

Note that you can't use this function to determine whether those columns were in the same record—the changes could have easily been made in separate records. SQL Server provides another clause that can be used to check multiple columns for modification in a single comparison: COLUMNS_UPDATED(). This function returns a varbinary bit pattern that indicates which columns in the table were updated. This function is well documented in the SQL Server Books Online, but I personally never use it because, well, because it returns a varbinary bit pattern with flags for the columns that were updated. If you like working with bit logic, it makes for an interesting study. Personally, I like to stick with the IF UPDATE statement and multiple tests because it's easier to read and maintain. If you modify the order or number of fields in your table, you can easily break code that depends on the COLUMNS_UPDATED function. In the case of an INSERT statement, either of these functions will specify that all columns have been updated.

Knowing whether a column or columns have been updated is only half the battle. The next challenge is determining exactly which records were affected. This can be done with some creative joins in your SQL statements. Assume for a moment that you're coding a trigger for all three data modification statements. In order to determine which records were affected by which type of modification, you can create joins using the following logic (you'll need to replace PKField in the following SQL statement with the primary key field or fields of your table):

- For an INSERT statement, the records that were inserted are those that appear in the inserted table but not in the deleted table:

```
SELECT * FROM inserted
LEFT JOIN deleted
  ON inserted.PKField = deleted.PKField
WHERE deleted.PKField IS NULL
```

- For a DELETE statement, the records that were deleted are those that appear in the deleted table but not in the inserted table:

```
SELECT * FROM deleted
LEFT JOIN inserted
  ON deleted.PKField = inserted.PKField
WHERE inserted.PKField IS NULL
```

- For an UPDATE statement, the records that were modified are those that appear in both tables. If you want to work with the data after the update, write the SELECT clause to retrieve from the inserted table. If you want to examine the original values in the table, your SELECT statement should retrieve from the deleted table:

```
SELECT * FROM [inserted|deleted]
INNER JOIN [deleted|inserted]
  ON inserted.PKField = deleted.PKField
```

Trigger execution

You may be wondering—when using multiple triggers that react to modifications to the same table—in what order your triggers will execute. The only general rule you can apply to this question is that INSTEAD OF triggers will fire before AFTER triggers. (Incidentally, if you have relationships defined that have cascade updates or deletes, the cascades will complete before any AFTER triggers fire.) If you write multiple INSTEAD OF triggers, you have no way of determining or setting the firing

Using Cursors in Triggers

Those of you who know how to create cursors in SQL Server may be tempted to use them to handle each record individually (for those of you who don't know what they are, cursors allow you to loop through individual records in a result set much like you would in ADO—record by record). Whatever you do, don't do this. Using cursors in triggers will result in a substantial degradation in the performance of your application. In my research leading up to this article, I was very disappointed to find a Microsoft publication written by the SQL Server development team that illustrated using a cursor in a trigger. The text then went on at least twice in the same chapter to say never to do this. I'll do one better: I won't show you how to do it so that you never will. Remember that cursors are evil! That's all you need to know about them when it comes to triggers.

order of any of these triggers. This means that when you code INSTEAD OF triggers, you must code each trigger so that it can stand alone and not expect it to depend on the action of another trigger.

When it comes to AFTER triggers, you have some—but not complete—control over the execution order of triggers. SQL Server includes a system stored procedure called sp_settriggerorder. This stored procedure allows you to define which trigger will fire first and which trigger will fire last for a particular data action on the host table. The syntax for calling this stored procedure is:

```
sp_settriggerorder[@triggername = ] 'triggername'
    , [@order = ] 'value'
    , [@stmttype = ] 'statement_type'
```

The first argument is the name of the trigger in question. The second argument can be one of three values: 'First', 'Last', or 'None'. Setting this argument to 'None' returns a trigger to its default behavior of being fired in an undefined order. The final argument is the statement type for which the trigger will fire. This can be 'UPDATE', 'INSERT', or 'DELETE'. Note that the trigger in question must be defined to be raised against the statement type set in this final argument, or the stored procedure will return an error.

Despite the existence of this feature of SQL Server, I would strongly recommend that you avoid using this stored procedure. Once you set this property, there's no way to determine that it's been set. This can lead to an extremely fragile database schema. A small change to the triggers by a developer who doesn't know that trigger execution order is important could have terrible consequences. If you need to ensure that triggers perform data modification in a series of steps, write a single trigger to handle all of the dependent events and code your logic into that trigger.

Triggering triggers

What happens when a trigger updates another table, or even its own table? Do the triggers on the destination table fire as well? If you make changes that would raise the active trigger again, will it fire? Well, it all depends. There are two concepts to understand when looking at triggers that raise other triggers: nesting and recursion.

Nesting of triggers is the concept that trigger trgA on table A could update data in table B, raising trigger trgB. The trigger on table B could in turn update data in table C firing trigger trgC. Whether triggers will raise other triggers is a server-wide setting. You can change the configuration of this setting using Enterprise Manager by selecting the server name in the Enterprise Manager tree and selecting Action | Properties from the menus. Select the Server Settings tab as shown in **Figure 1**.

By checking or unchecking the second item under the Server behavior section, you can define whether triggers will fire other triggers. The setting is on by default in SQL Server 2000.

Trigger recursion is the concept that a trigger can raise itself. There are two types of recursion: indirect and direct. Indirect recursion involves a trigger raising another trigger that results in other triggers firing that eventually lead back to the same trigger being fired again. This type of recursion is considered to be a manifestation of nested triggers, and so it's controlled by the server-wide nested trigger setting.

Direct recursion involves a trigger updating data in its own table such that the same trigger will be raised again. Direct recursion can be allowed or prevented by changing a database-wide setting. Select a database in Enterprise Manager and choose Action | Properties from the menus. The Options tab of the Database Properties dialog is shown in **Figure 2**.

The second item down in the Settings section on this tab controls whether or not triggers can cause direct recursion. The default setting for this option is off.

Leaving these settings at their default values is normally the best situation. However, there are occasions where you might consider changing them. You might be building an inventory management system that has a trigger to update the quantity of an item in the Inventory table whenever some of the item is ordered. This action

Figure 1. Server properties.

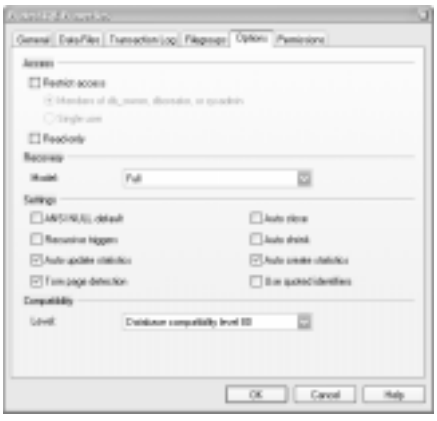


Figure 2. Database properties.

can easily be handled by a trigger defined on the table when a row is added to the OrderLineItems table.

This inventory system may also require that directly updating the Inventory table (for instance, when inventory is checked in an audit) creates backed-up corrective entries somewhere else in the database. If this is the case, you wouldn't want the trigger that modifies the quantity on-hand (due to an order being placed) raising the trigger that creates the corrective entry. You might end up with an infinite loop. In this case, you may want to disable nested triggers.

Conversely, suppose that you're building a financial system with a list of companies that lets you define what companies own other companies. You might have a field that specifies the number of holdings (total child companies) that a particular company has. This could be updated by a trigger that simply counts the children of that company when a new child company is added. If you want this total to include all of the grandchildren, great-grandchildren, and so on, you could turn on recursive triggers to have this number cascaded up the line. The new values would be the count of all children plus the total of all children's children.

Although you can change these options—and there are times when you might consider it—I recommend that you leave them as they are at their default values. Although you think a change might make one situation easier to deal with, you'll inevitably cause problems in other areas. The same effect can usually be programmed into a trigger or stored procedure in the special cases where it's required.

Canceling triggered events

One of the strengths of triggers lies in their ability to intercept an action that was or will be performed on the data. You may use a trigger to validate data going into the database in cases where constraints just can't do the job. Of course, if the change fails your test you'll want to cancel the update that triggered your trigger. You can cancel the change in one of two ways. In INSTEAD OF triggers, the solution is simple—do nothing. Since INSTEAD OF triggers react to the event before it happens, you must explicitly define what action is taken after the trigger is fired. If you don't specify any action, no data will change.

In the case of AFTER triggers, it's important to understand that any action that takes place in this type of trigger automatically belongs to a transaction. Because of this, you can simply roll back the transaction to cancel the event and any other modifications you made in the trigger. This can be done in T-SQL with this code:

```
ROLLBACK TRANSACTION
```

If the action that caused the data modification was itself running in a transaction, the entire outer transaction will also be rolled back by this call.

Trigger limitations

As with all great things, even triggers have their limitations. Here are some of the more important ones you need to keep in mind when designing triggers:

- Trigger nesting can only occur up to 32 levels. After this, SQL Server raises an error and automatically rolls back the triggering action. This prevents infinite loops in nested or recursive triggers.
- Only one INSTEAD OF trigger can be defined for each triggering action (UPDATE, INSERT, or DELETE).
- INSTEAD OF triggers never fire in direct recursion.
- INSTEAD OF triggers can't be defined for DELETE or UPDATE actions if the table has a foreign key with cascade settings turned on for the respective action on the parent table.
- INSTEAD OF triggers affecting text, ntext, or image fields have special considerations. See the SQL Server Books Online under "Using text, ntext, and image Data in INSTEAD OF Triggers."

Where's the code?

So, you made it this far, and you're probably wondering where I put the sample code. Sorry to say, but there isn't any this month. I have a very good reason for this. Any code that I write will be written in Transact-SQL. Although this series is called "Working T-SQL," I have a big consideration in mind. A new version of SQL Server is headed our way, currently code-named Yukon (probably to be released as SQL Server .NET). This new version of SQL Server will allow you to program SQL Server objects in any .NET-compliant language. As such, it's important that you understand the concepts of how triggers work so that you can use this information in the next version of SQL Server.

Don't despair—next month you'll see another article that shows you how to apply this knowledge with the current version of SQL Server. I'll demonstrate much of what I've talked about with some real code and show you how to create and maintain SQL Server triggers in Access. And I'll show you how you can use them to take some of the code out of Access and make your application more reliable. ▲

Russell Sinclair is an MSCD and the owner of Synthesystems, a technology consulting firm specializing in Visual Basic, SQL Server, and Microsoft Access development. He's the author of *From Access to SQL Server*—an Access developer's guide to migrating to SQL Server. He's a senior programmer with Questica, Inc., a company specializing in software for custom-design manufacturers, and is a *Smart Access* Contributing Editor. www.synthesystems.com, russell@synthesystems.com.

CompareWiz 2002

Danny J. Lesandrini

97 2000 2002

This month's Product Review features an inexpensive Add-In that could save you hours of development time. Danny J. Lesandrini explores this simple, yet powerful tool.

OCCAM'S (or Ockham's) razor is a principle attributed to the 14th century logician and Franciscan friar William of Ockham. It simply states, "Entities should not be multiplied unnecessarily." Occam's razor is sometimes cited in stronger forms than he intended, and it's the following version that often guides my software development practices: "If you have two equally likely solutions to a problem, pick the simplest."

There are plenty of complicated tools on the market (with lots of bells and whistles) that will compare Access databases. While there's nothing wrong with having a rich feature set, sometimes the simplest solution is the best. CompareWiz 2002, marketed by Software Add-Ins of Sydney, Australia, passes the Occam's razor test for simplicity. A single-user license sells for only $45 USD, making this a very affordable developer tool. You can try it out for yourself by downloading a demo copy for free from www.softwareaddins.com. Though the demo copy imposes some limitations on what may be compared, it'll give you a clear idea of what this tool can accomplish.

There are times, when working with multiple copies of a database, that you may lose track of the precise changes that you've implemented. I remember times when some small, insignificant change created an undesirable effect (I guess you'd call that a bug) that was almost impossible to locate. Comparing a previously working copy of the application to the broken one was all that I needed to do to quickly identify the offending change.

Bug tracking isn't the only reason to have a comparison tool. Some additional ways that CompareWiz 2002 can help you include:

- Compare differences between two copies of the same database.
- Find and report all undocumented developer changes.
- Determine which database copy has your last set of changes.
- Synchronize development and client database copies.
- Track and document differences in the different development versions of an application.

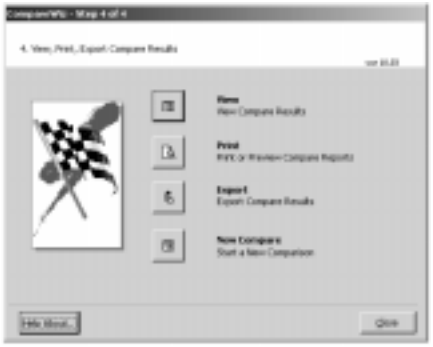


Figure 1. The wizard's compare results options screen.

How it works

CompareWiz 2002 is a Microsoft Access Add-In, so after installation it's available on the Tools | Add-Ins menu. Selecting CompareWiz 2002 launches a four-step wizard that walks you through the process of choosing compare options. The first step is to identify another Access mdb file to compare with the currently opened database. Next, you can elect to do one of the following:

- Compare all database objects (tables, forms, queries, and so forth).
- Customize your selection to a specific subset of objects.

The wizard's next step allows you to pick which properties you wish to include or exclude from the comparison. Clicking the Finish button begins the compare process, which, in my tests, took about 20 minutes to analyze all the objects in a medium-sized database application.

Without going into great detail about the actual wizard interface, it's enough to say that it's very intuitive. I never had to open a Help file to understand the choices presented to me, and the process flow was natural and comfortable. The first time I ran CompareWiz 2002, I selected all database objects and all properties, but upon examining the results, it was clear that to benefit from this tool I needed to filter for the things that I really wanted to compare.

Viewing the results

The last step of the wizard, which appears after all analysis has been completed, is shown in **Figure 1**. From this screen you may elect to export the results to a file, print, or view them in any of a number of different formats. Once again, simplicity is the key here. The View option opens an Access form that allows you to browse the results. A set of option buttons at the bottom of the form allows for quick filtering for a specific type of entry (say, query differences). Double-clicking a line item opens a window with more detail about the comparison

difference (see **Figure 2**).

The Print option exposes a number of prepared comparison reports for previewing and/or printing. Again, these reports are simple—exposing all differences for each particular type of database object, such as tables, forms, queries, and so on. While the reports are simple, they may be all that you need. The Export function allows you to output the analysis results to a file in a variety of formats, including text, HTML, and XML. Exporting the data is a great way to document changes between versions of a distributed application. Keeping these copies of your changes lists provides a history of the application's evolution.

Figure 2. Browse and view screen for compare results.

Conclusion

One annoying aspect of the compare process was that each time a database object such as a form or report was analyzed, its window stole the focus. What this means is that while CompareWiz 2002 is analyzing your database, you can't do any other kind of work on your computer. I attempted to read my e-mail, but each time I'd click on my Outlook Inbox, Access would open a new form that suddenly became the active window, pushing Outlook into the background. Fortunately, as I mentioned, the process took only about 20 minutes, and the annoyance occupied only about half of that time.

In spite of this drawback, CompareWiz 2002 is a convenient, inexpensive little tool that's worth a look. While you're at the SAI site, check out their other products: ReplaceWiz, and CompareDataWiz. Although this review was based on the Access 2002 version of the Add-In, it's noteworthy that all of these products are available for Access 97 and Access 2000 versions as well. Remember Occam's razor: All things considered, go with the simple solution. ▲

Danny J. Lesandrini, a Microsoft Certified Professional in Access, Visual Basic, and SQL Server, has been programming with Microsoft Access since 1995. He maintains a Web site containing Access-related code solutions at http://datafast.cjb.net. datafast@attbi.com.

Dates, Data Access, and Presentation

Peter Vogel

This month, Peter Vogel looks at a problem in managing dates and displaying information using conditional formatting. He starts with a solution to the problem, but uses that as a springboard to discuss what processing should be done in the different parts of your application.

THIS month's column addresses a single question in two parts. The first part of the question revolves around how to retrieve the data, while the second part discusses how to display the result. My answer here is almost certainly overkill. What I want to demonstrate in this column are the kinds of issues that you should be considering when answering even the simplest problems.

My problem is that I've got a table with a set of scheduled activities. Each table has a startDate field and an endDate field. What I want to do is find every activity that either begins or ends within a specific date range. In addition, sometimes the date range that I'm searching in is only one day long. Also, sometimes my scheduled activities are only a day long. I need to flag those activities that either start or end outside the date range.

I'm going to give you a SQL statement that will retrieve the data that you need. You can do the following:

- Use the statement in a query that's the RecordSource for a form or report.
- Insert the resulting data into a table and display it in a form or report.
- Set a form's Recordset property equal to the recordset returned by the SQL statement (if you're using Access 2000/2002).

Once the data is returned, I'm going to use forms-based processing to handle displaying the data. In other words, I'm distinguishing between the business/data-related activities (the SQL statement) and the presentation layer activities.

The first step in the SQL statement might be to look for all activities that either begin or end within the period. So, assuming that 01/01/2001 and 12/31/2001 are your test dates, you might write a SQL statement like this:

```
Select *
    From tblScheduledItems
    Where startDate > #01/01/2001# or
          endDate < #12/31/2001#
```

The problem is that this SQL statement will return every record in the table. The statement will find all items that started after 01/01/2001—even if those items start after the period's end date of 12/31/2001—plus all the items that started before your end date—even if they ended before your period's start date. To handle this, this slightly more complicated SQL statement should work:

```
Select *
    From tblScheduledItems
    Where (startDate Between #01/01/2001# And
          #12/31/2001#)  Or
            (endDate Between #01/01/2001# And
          #12/31/2001#)
```

This does the job and (since the Between statement is inclusive) even includes scheduled activities that end on 01/01/2001 or begin on 12/31/2001. Furthermore, if you have an index that consists of just the startDate and the endDate, the Between operator should trigger the optimizer to use that index to speed up your search. A quick test with a start and end date of 05/21/2001 (that is, a single day's date range) shows that this code also catches activities that start and end on a single day.

It's also possible to flag the items that start or finish outside the date range using SQL. The following solution uses three SQL statements to find the following items:

- Those that start in the period (but don't end in it).
- Those that end in the period (but don't start in it).
- Those that both start and end in the period.

To put all the results in a single recordset, I've used the Union operator to join the three statements (the Flag pseudo-field indicates which scheduled items are completely within the period):

```
Select *, 0 As Flag
    From tblScheduledItems
    Where (startDate Between #01/01/2001# And
          #12/31/2001#) And
           (endDate > #12/31/2001#)
Union
Select *, 0 As Flag
    From tblScheduledItems
```

```
    Where (endDate Between #01/01/2001# And
           #12/31/2001#) And
            (startDate < #01/01/2001#)
Union
Select *, 1 As Flag
    From tblScheduledItems
    Where (startDate Between #01/01/2001# And
           #12/31/2001#) And
            (endDate Between #01/01/2001# And
           #12/31/2001#)
```

As fond as I am of "pure SQL" solutions, I suspect that most SQL parsers will process this as three separate statements, tripling your runtime. In addition, because some of the tests only use the endDate, a single index on startDate and endDate probably won't be helpful in speeding up processing—you'll need a separate index on each of those two fields. However, the benefit of a solution written this way is that, should you want to add a new classification, you may be able to get away with just using a Union statement to add another Select statement.

Let's take one more stab at a pure SQL solution. By using the IIf function, I can set the Flag variable depending on the record's start and end dates:

```
Select *,
   IIf(startDate > #01/01/2001# and
      endDate < #12/31/2001#,1,0) As Flag
    From tblScheduledItems
    Where (startDate Between #01/01/2001# And
           #12/31/2001#) Or
            (endDate Between #01/01/2001# And
           #12/31/2001#)
```

I could also use a VBA function in my SQL statement. Using a VBA function, however, highlights one of the problems that I have with this solution.

First, there's a potential performance issue. Obviously, your relational database isn't going to execute a VBA function that you have in your MDB file. In order to execute this function, Access would have to switch back and forth between VBA processing and processing done by the database engine. No matter how cleverly this is done, it's going to be a problem. Using an IIf statement assumes that the back-end database supports that statement. If the back-end database doesn't support the IIf, Jet will step in and supply the functionality—and then you've got the same problem as you would with VBA.

Second, and more importantly from my point of view, once you start thinking about writing code like this, it seems to me that you've moved out of the world of data access and into the world of business rules and presentation logic. I'd prefer to keep those worlds separate. I wouldn't be surprised, for instance, if the next requirement that you get for this data is to flag all tasks that start and end within the time period. In fact, there might be an infinite number of ways that this data might be presented, and I don't want to build those presentation choices into my data access code.

Besides, what would these SQL solutions save you? At best some recordset processing that will do some simple compares on a set of records held in memory. There isn't a lot of time to be saved here. Furthermore, by putting the logic in the SQL statement, you're transferring the work to the database server that's shared by all the users. By keeping the logic in your form, you're creating a distributed application where the application's processing is shared among many computers (in this case, the user's computer).

Here's my next problem: When I highlight the Activity Name field for the projects that don't end in the project (by setting the text box's BackColor property to red), all of the records in my continuous form get highlighted in red. How can I highlight the field on just one record?

This question crops up frequently and is actually a good introduction to object-oriented programming. When you change the BackColor of the text box, you're actually changing the definition of that text box. So, when the text box is repeated on a continuous form, the text box repeats with its new BackColor. Changing one text box changes the definition of every instance of the text box.

Many developers' initial reaction is annoyance when they find that every text box has a BackColor of red when

they only wanted to change one. After all, each text box is displaying a different field value, so why can't each text box display a different BackColor? However, when you think about it, each text box is displaying the same value: whatever the value is for the field that the text box is bound to based on the current record. If the text box has its ControlSource property set to a field name (for instance, ActivityName), every instance of that text box will be bound to that field. On an instance-by-instance basis, that ControlSource generates a different value, but the ControlSource setting is the same for all instances of the text box.

With Access 2000 and later, you can use conditional formatting to control the appearance of individual instances of the text box. To work with this feature you select a text box (or combo box, which consists of a text box and a list box) from the Access user interface, and from the Format menu select Conditional Formatting. That brings up the dialog box shown in **Figure 1**. In this dialog you can set the conditions that control how the text box is formatted. You can have up to three different conditions on any text box. For instance, in **Figure 2** you can see that I'm setting the BackColor of the startDate when it's before the start of the period, and the BackColor of the endDate when it's after the end of the period.

What you're doing with the Conditional Formatting dialog is managing the FormatCondition objects associated with the text box. You're altering the definition of the control by creating FormatCondition objects that will be associated with the control. The same collection of FormatCondition objects will be applied to all instances of the text box. However, on an instance-by-instance basis, those objects produce different results.

In your original request, you wanted to highlight the activity name. Unfortunately, you can't format one field based on the value in another field. If you try to call a function, you'll find that you won't be able to access the values in the other text boxes on the row. This also means that you can't set the conditional formatting based on the results of two fields (something like flagging the activity name based on the start date and the end date). Instead, what I've done is flag the end date if it's past the end of the test period, and the start date if it's before the start of the test period. As a result, items that are completely within the period aren't flagged at all.

In your case, you also need to set the FormatCondition objects dynamically at runtime, since the criteria for flagging the fields depends on what start and end dates were used to select the results. That works out well for me, because the code more clearly shows what's happening under the hood as new FormatConditions are added to the text, changing the definition of the text box:

```
Dim fcd As FormatCondition

Set fcd = Me.startDate.FormatConditions.Add( _
        acFieldValue, acLessThan, "#1/1/2001#")
fcd.BackColor = vbRed

Set fcd = Me.endDate.FormatConditions.Add( _
        acFieldValue, acGreaterThan, "#12/31/2001#")
fcd.BackColor = vbRed
```

When I'm teaching a class, by the third day everyone begins their questions with "I have a short question..." Unfortunately, as I demonstrated, I don't have any short answers. The point here, however, isn't in the answer but in the way we got there. As an Access developer, these are the kinds of decisions that you should be making with every line of code that you write. ▲

DOWNLOAD **AA0103.ZIP at www.smartaccessnewsletter.com**

Peter Vogel (MBA, MCSD) is the editor of *Smart Access* and a principal in PH&V Information Services. peter.vogel@phvis.com.

Figure 1. The Conditional Formatting dialog.

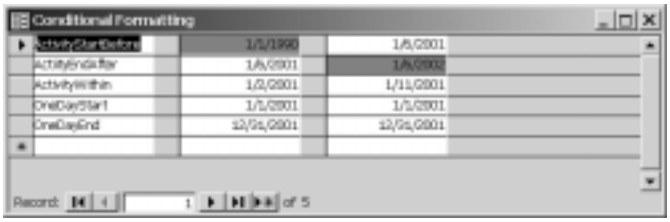


Figure 2. Conditionally formatted text boxes.

Know a clever shortcut? Have an idea for an article for *Smart Access*?
See the back page for the contact information where you can send your ideas.

Recursion, Part 1...

Continued from page 8

After this automated examination, you might have entries that describe subforms that are dynamically loaded in a tab control's OnChange event. Several are shown in Figure 1. Being dynamically loaded, the SourceObject property will be blank and thus, NavigationMap can only pass pstrNameOfObjectToExamine to LogItAndDrillIn. There's also a special circumstance when the subform / subreport source object can already be opened in a previous call. In this case, NavigationMap() passes "UNABLE TO OPEN" & pstrNameOfObjectToExamine to LogItAndDrillIn(). A third type of entry occurs for parameterized calls to OpenForm / OpenReport methods. These you'll have to trace by hand and make manual entries for tracing.

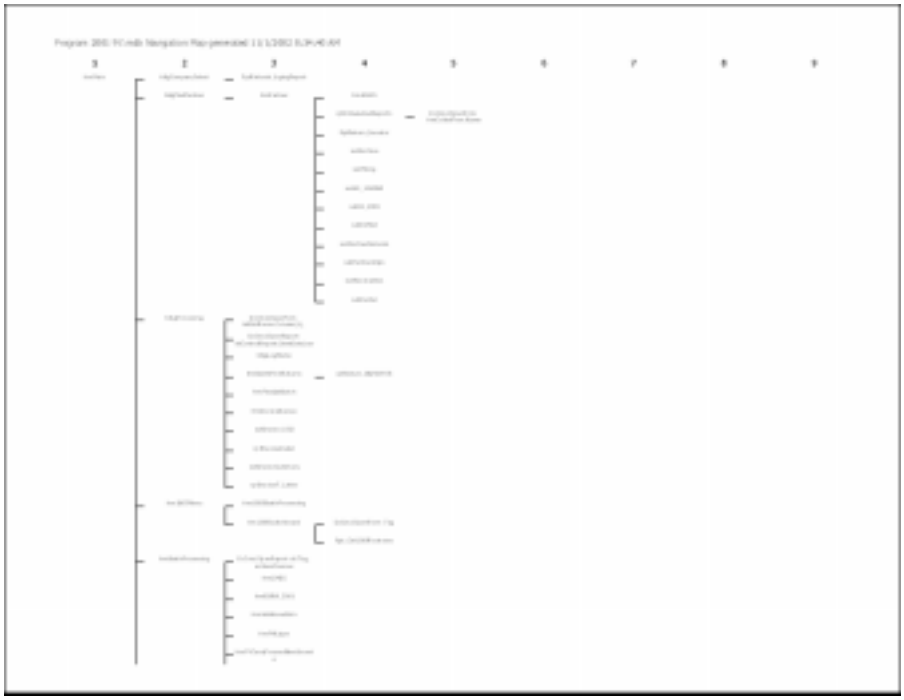


Figure 6. I'll talk about zsrptNavigatonMap next month.

Because I always use the LR naming convention for my subform controls, it's usually easy to determine that a control named "subXXXX" is a container for "fsubXXXX". So, for the subform entries, I use the Find and Replace dialog to replace entries that begin with "UNABLE TO OPEN" with the letter "f", and entries that begin with "sub" with "fsub" (see **Figure 5**). I then press the button Load Current Row for each entry and then the Map It button. The dissociated subform opens and is examined just as if it were found.

The last two buttons at the top right of the main form allow you to save the map you've generated as "zstblNavigationMap" concatenated to Now, and run the report that displays the tree diagram resulting from recursive analysis of the navigation map data (**Figure 6**).

The query for this report and the formatting of the lines that connect the nodes is another story that I'll save for next month's issue.

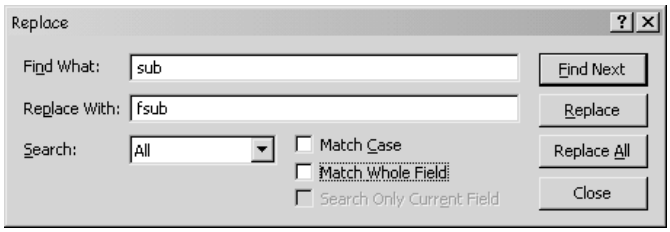


Figure 5. Replacing dynamically loaded subform entries with the name of the subform.

Do try this at home

Remember, nothing's perfect. So far, my Navigation Mapper has done a pretty decent job on every database

I've tested, but I really have no way of testing whether this thing absolutely works every time. Most of the errors I've encountered have been unexpected holes in my logic. Personally, my clients and I are happy just to have the roadmap it creates as a starting point, though usually it's complete.

Until next month, explore the objects in the accompanying database by using the Load Switchboard and Map It buttons. The different problems discussed in this article—including hyperlinks, custom OpenAForm() and OpenAReport() functions, and even a dynamically loaded subform—are included in the sample database. The database has no data behind it, just a series of blank forms and reports, but you'll get the idea.

To use my Navigation Mapper in your own database, copy zstblNavigationMap, zsfrmNavigationMap, and zsfsubNavigationMap to a backup of your file. You'll have to turn on Hidden Objects in the Tools | Options dialog to import and use them. If you used the Switchboard Manager to create your opening form, just press the Load Switchboard button. If you have a custom form, enter 1, *formName*, *formName*, and select Form as the type of object called before clicking the Map It button. Then watch the fun begin. Any unconfirmed entries will need your attention, but hey, it beats going in cold on that new project. ▲

NAVIGATE.ZIP at www.smartaccessnewsletter.com

Christopher Weber travels throughout the country teaching Access development and programming seminars for The DSW Group in Atlanta, GA. He's been an Access developer since its first release, enjoys working with clients, and heads the DSW Group's Access development and training team. www.access-training.com, cweber@thedswgroup.com.

January 2003 Downloads

- NAVIGATE.ZIP—The sample database from Christopher Weber provides a set of routines for generating a table that records your application's navigation structure in a table. The form included allows you to review the results and update it. (Access 97 and 2000)
- AA0103.ZIP—Peter Vogel's sample database provides a technique for extracting date ranges that leverages an index on the database to ensure the fastest possible performance. In the sample form, Peter shows how (in the presentation layer) you can use conditional formatting to highlight entries in a list. (Access 2000)

For access to all current and archive content and source code, log in at www.smartaccessnewsletter.com with your unique subscriber user name and password. For access to this issue's Downloads only, click on the "Source Code" button, select the file(s) you want from this issue, and enter the User name and Password at right when prompted.

User name `airplane`

Password `ballet`

Editor: Peter Vogel (peter.vogel@phvis.com)
Contributing Editors: Mike Gunderloy, Danny J. Lesandrini, Garry Robinson, Russell Sinclair
CEO & Publisher: Mark Ragan
Group Publisher: Connie Austin
Executive Editor: Farion Grove
Production Editor: Andrew McMillan

Questions?

Customer Service:
Phone: 800-493-4867 x.4209 or 312-960-4100
Fax: 312-960-4106
Email: CService@Ragan.com

Editorial: FarionG@Ragan.com

Advertising: HowardF@Ragan.com

Pinnacle Web Site: www.pinnaclepublishing.com

Subscription rates

United States: One year (12 issues): $169; two years (24 issues): $287
Canada:* One year: $189; two years: $321
Other:* One year: $194; two years: $330

Single issue rate:
$20 ($22.50 in Canada; $25 outside North America)*

* Funds must be in U.S. currency.

Smart Access (ISSN 1066-7911)
is published monthly (12 times per year) by:

Pinnacle
A division of Ragan Communications, Inc.
316 N. Michigan Ave., Suite 400
Chicago, IL 60601

POSTMASTER: Send address changes to Lawrence Ragan Communications, Inc., 316 N. Michigan Ave., Suite 400, Chicago, IL 60601.