

This article is an excerpt from [Building Microsoft Access Applications](#), from Microsoft Press (ISBN 0-7356-2039-3, copyright Microsoft Press 2005, all rights reserved). The author of the book is John Viescas.

John L. Viescas is the author of Microsoft Office Access 2003 Inside Out and the popular Running Microsoft Access books from Microsoft Press. He is president of Viescas Consulting, Inc., a respected provider of database application design and editorial consulting services. He has been recognized by Microsoft Product Support Services as a Most Valuable Professional (MVP) every year since 1993 for his contributions to the community of Access users.

No part of this chapter may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, electrostatic, mechanical, photocopying, recording, or otherwise—without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

3

Designing A Client/Server Application

Unless you're building an application for your own use or creating a specialized application for a single user, you need to understand the best ways to allow multiple users to share your application. One of the key reasons developers move an application from Excel to Access is the need to allow multiple users to share and update the same data at the same time. With Excel, unless you have marked the workbook for sharing, only one user can have the workbook file open for editing at a time. Even when you share a workbook, Excel either saves the last change to a cell without warning or asks the user to resolve the conflict. But database systems are designed to share the data, and multiple users can be editing data in the same table at the same time.

In this chapter, you'll explore the various client/server architectures you can employ. You'll also learn how to design your application to minimize network resources and conflicts. Finally, you'll learn about advanced techniques that you can

use to allow your application to be shared by 50 or more users at the same time—even when you're still using the JET desktop engine to manage the data.

Understanding Access Client/Server Architectures

To allow multiple users to run your application at the same time and share the data, you must make the data available on a server accessible to all the users. The biggest mistake you can make when you want multiple users to run your application is to place a single desktop database file (.mdb) on a server file share. When you do that, a copy of Access must still run on each desktop, and each user's copy must fetch not only the data but also the query, form, report, and module definitions over the network. This additional traffic over the network slows down not only your application but also any other applications or tasks needing to send data over the network (such as sending a report to a network printer).

Also, because some user actions (such as applying a filter to a form) actually change a form definition, Access must be able to periodically save a form back to the shared database. When that happens, Access on the user's machine must be able to temporarily acquire exclusive access to the object. If your application needs to create temporary objects such as a working table or query, a user will see a locking error if another user is in the same part of the application at the same time. If any error occurs on the network while Access is trying to save an object, your shared database can become corrupt.

You can, however, design the architecture of your Access application to provide efficient sharing of data. Figure 3-1 shows you the most common client/server architectures for an Access application.

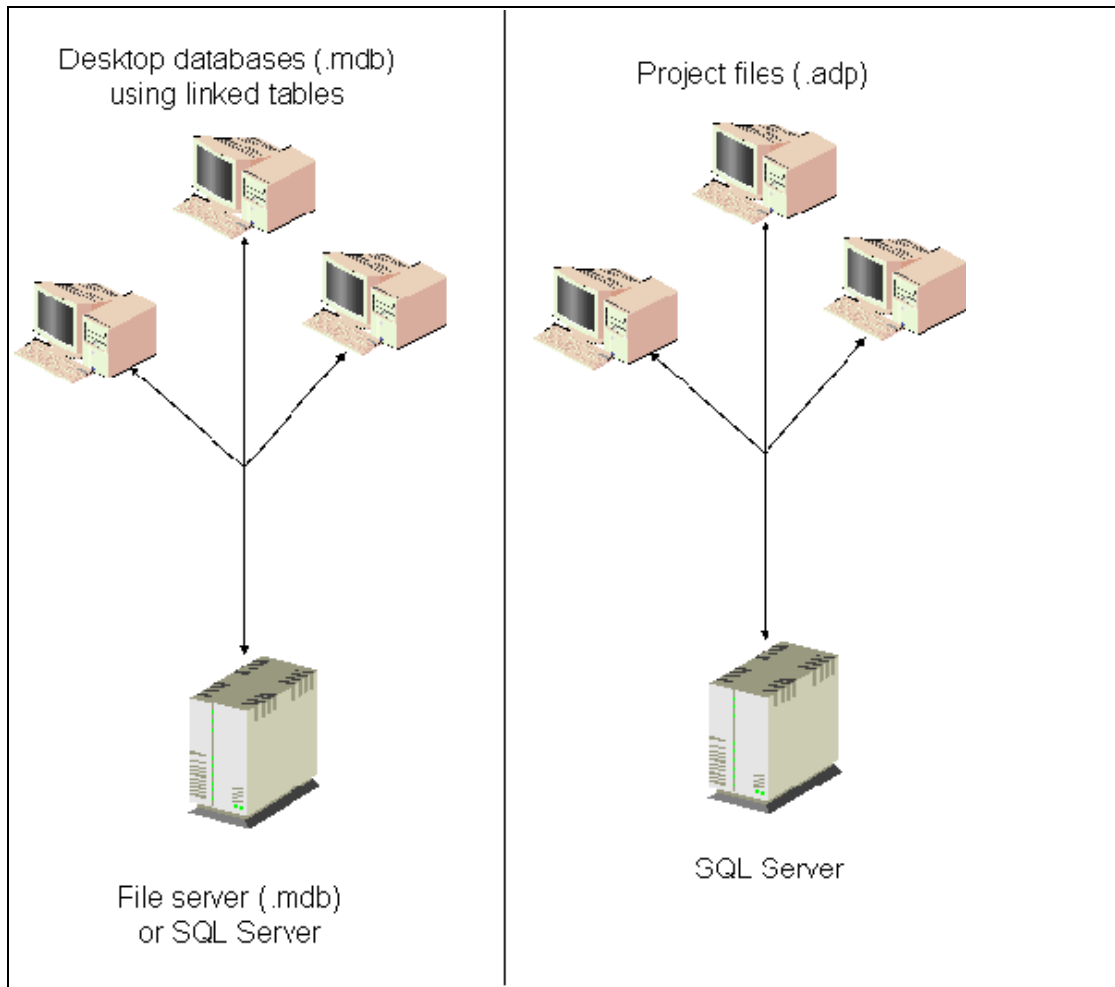


Figure 3-1.
Client/server architectures for an Access application.

The most commonly used and simplest to develop architecture, shown on the left of Figure 3-1, involves giving each user a copy of a desktop database file (.mdb or .mde) that contains only local tables and the definitions for the queries, forms, reports, and modules that define your application. This desktop database file uses

linked tables to connect to the shared data residing in either another desktop database file (.mdb) or in SQL Server.

Keep in mind that Access running with the JET database engine is fundamentally a desktop database. Although JET provides many data management capabilities similar to what you'll find in a full-fledged database system such as SQL Server, it's really only a very smart file manager that understands SQL. So, all requests for data execute on the client machine, and the JET engine makes low-level file requests to the file system. When the data is in a shared file on a file server, those requests must be sent over the network, and the necessary physical file pages must be sent by the server to the desktop machine for further processing by the JET engine. This is in sharp contrast to a true database system like SQL Server where the client sends an SQL command, the server finds the data, and the client receives only the requested rows from the server.

Access 2000 introduced a new type of application file—an Access project (.adp) shown on the right side of Figure 3-1—that allows your application to connect directly to an SQL Server database. Access 2002 greatly improved the stability and features available in project files. You can design SQL Server tables directly from Access, and you can design and work with SQL Server views, stored procedures, and functions defined in the SQL Server database. You should certainly consider using this architecture if your application needs to manage very large amounts of data or needs to support more than 20 concurrent users.

Because you're working directly with SQL Server with a project file, communication with the server is potentially more efficient because all the work to fetch and update data is done by the server. However, designing an application to work with SQL Server is more complex. For example, you must be careful to assign a unique name to any temporary table that you create because all data is shared—you cannot access or create tables on the user's local machine. And you can still create a very inefficient application if you do not employ techniques to minimize the amount of information transferred over the network.

Programming for Client/Server Architecture

Regardless of whether you're using a desktop database file (.mdb) on a file server or SQL Server to manage your data, you should design your application to request only the data needed for each specific task. If, for example, you design forms that always open to display all available records in the table you want to edit, you are transmitting far more data over the network than might be needed for a specific task. It's the network that is the slowest component in a client/server application, so the less information that your application transmits over the network, the better your application will run. The following sections describe techniques that you can implement in any client/server architecture to accomplish the goal of transmitting only the data absolutely needed for each task.

Note: All the sample applications included with this book use the simpler client/server architecture that stores the data in a desktop database file (.mdb) on a file server and uses linked tables in the application code running on the client. When an application using this architecture is well designed, it can easily be
--

upsized to an Access project file (.adp) and SQL Server. For details about upsizing a desktop application to a project, see Chapter 24 in *Microsoft Office Access 2003 Inside Out* (Microsoft Press, 2003).

Filtering Forms and Reports

When you first start building applications using Access, it's very simple to select a table you want to edit and run the Form Wizard to create a form that edits all the data. As you become a bit more experienced, you might create a query first that includes data from related tables (such as Customer information for an Order) and then build your form. These techniques work fine for a single-user application, but they are potentially disastrous when you're editing data stored on a server and your application is shared by multiple users.

Remember that the key to designing an efficient client/server application is minimizing the amount of data sent to or fetched from the server. When your forms always open displaying all the available data and the user needs to work with only one or two records, you're asking Access and the database system to do far more work than is necessary. The same is true when you open a report that needs to display or print only a few records. If your tables contain tens of thousands of rows, your application performance will be terrible.

The best way to focus the user on the task at hand and fetch only the rows the user needs is to force the user to enter some criteria that specifies or narrows down what records are needed. In all the sample applications, you'll find that you cannot

click a button on the main application switchboard to open a form editing all records.

For example, when you click the Orders button on the main switchboard in the Inventory Management sample application, the application displays a pop-up filter window that forces you to narrow your choices, as shown in Figure 3-2.

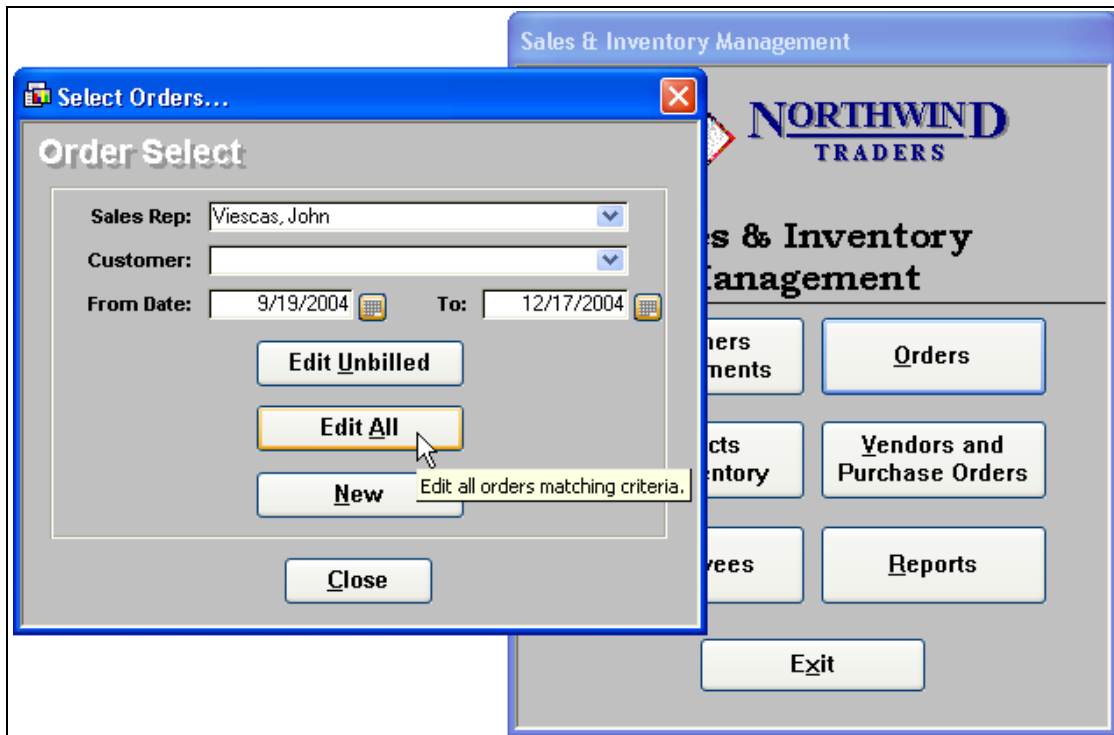


Figure 3-2.

The Inventory sample application forces you to specify filtering criteria for Orders.

Notice that the form automatically fills in the name of the currently signed-on sales representative. If you open the Customer list, you'll find that the form has also filtered the list of customers to only those assigned to the current sales representative. When I designed the form, I set the Default Value property of the From Date text box control to `=Date()<=Date()+90`, and the Default Value property of the To text box control to `=Date()`, but the user can set a wider or narrower range if necessary. If the user tries to clear all the criteria and then click Edit Unbilled or Edit All, the

application code refuses to proceed. The application also requires the user to select a customer before clicking the New button, and it opens frmOrders in Data Entry mode, thus avoiding fetching any records from the database at all.

Here's the code behind the Edit All button that builds and tests the filter:

```
Private Sub cmdAll_Click()  
Dim strWhere As String  
    ' Open all orders for the selected customer  
    ' Set an error trap  
On Error GoTo cmdAll_Err  
    ' Call the predicate builder  
strWhere = BuildPredicate()  
    ' If returned nothing, then done  
If strWhere = "" Then Exit Sub  
    ' Close the form if it is open  
If IsFormLoaded("frmOrders") Then  
    If vbYes = MsgBox("The Orders window is already open. " & _  
        "This search will cancel any pending edits in that " & _  
        "window, close it, and " & _  
        "attempt to reopen with the criteria you specified." & _  
        vbCrLf & vbCrLf & "Are you sure you want to proceed?", _  
        vbQuestion + vbYesNo + vbDefaultButton2, gstrAppTitle) Then  
        ' Close using the form's Cancel routine  
        Forms!frmOrders.cmdCancel_Click  
    Else  
        Exit Sub  
    End If  
End If  
    ' Open frmOrders hidden to check if any records found  
DoCmd.OpenForm FormName:="frmOrders", WhereCondition:=strWhere, _  
    WindowMode:=acHidden  
If Forms!frmOrders.RecordsetClone.RecordCount = 0 Then  
    ' No orders for selected customers
```

```
        MsgBox "There are no orders for the " & _  
            "customer criteria you specified.", _  
            vbInformation, gstrAppTitle  
        DoCmd.Close acForm, "frmOrders"  
        Exit Sub  
    Else  
        Forms!frmOrders.Visible = True  
    End If  
    ' Close myself  
    DoCmd.Close acForm, Me.Name  
  
cmdAll_Exit:  
    Exit Sub  
  
cmdAll_Err:  
    ErrorLog Me.Name & "_cmdAll", Err, Error  
    MsgBox "Unexpected error: " & Err & ", " & Error, _  
        vbCritical, gstrAppTitle  
    Resume cmdAll_Exit  
End Sub
```

And here's the *BuildPredicate* function (that you can find in the form's module) called by the Edit All command button code:

```
Private Function BuildPredicate() As String  
    Dim varWhere As Variant  
    ' Initialize to Null  
    varWhere = Null  
    ' If specified a company value  
    If Not IsNothing(Me.cmbCustomerID) Then  
        ' .. build the predicate  
        varWhere = "CustomerID = " & Me.cmbCustomerID  
    End If  
    ' Do dates next  
    If Not IsNothing(Me.txtFromDate) Then  
        If Not IsDate(Me.txtFromDate) Then
```

```
        MsgBox "You must enter a valid 'From' date.", _
            vbExclamation, gstrAppTitle
        Me.txtFromDate.SetFocus
        Exit Function
    End If

    ' .. build the predicate
    ' Note: taking advantage of Null propogation
    ' so we don't have to test for any previous predicate
    varWhere = (varWhere + " AND ") & "[OrderDate] >= #" & _
        Format(Me.txtFromDate, "dd mmm yyyy") & "#"
End If

If Not IsNothing(Me.txtToDate) Then
    If Not IsDate(Me.txtToDate) Then
        MsgBox "You must enter a valid 'To' date.", _
            vbExclamation, gstrAppTitle
        Me.txtToDate.SetFocus
        Exit Function
    End If

    ' Make sure To is later or equal to From
    If Not IsNothing(Me.txtFromDate) Then
        If Me.txtToDate < Me.txtFromDate Then
            MsgBox "'To' Date must not be earlier than 'From' Date.", _
                vbExclamation, gstrAppTitle
            Me.txtToDate.SetFocus
            Exit Function
        End If
    End If

    ' .. build the predicate
    ' Note: taking advantage of Null propogation
    ' so we don't have to test for any previous predicate
    varWhere = (varWhere + " AND ") & "[OrderDate] <= #" & _
        Format(Me.txtToDate, "dd mmm yyyy") & "#"
End If

' Finally, do Sales Rep
If Not IsNothing(Me.cmbSalesRep) Then
    ' .. build the predicate
```

```
varWhere = (varWhere + " AND ") & "[SalesRepID] = " & Me.cmbSalesRep  
End If  
  
' Check to see that we built a filter  
If IsNothing(varWhere) Then  
    MsgBox "You must enter at least one search criteria.", _  
        vbInformation, gstrAppTitle  
    Exit Function  
End If  
  
' Return the filter string  
BuildPredicate = varWhere  
End Function
```

NOTE:

Notice that this sample code uses the WhereCondition parameter of the OpenForm command. This technique works equally as well in an Access project (.adp) as long as the form you are opening uses a table or a view as its record source. Access sets the form's ServerFilter property using the filter you specify, which causes the form to send the filter to the server. If the form uses a stored procedure or function as its record source, Access must send a command to the server to execute the procedure, and the procedure returns all the rows unfiltered. Access must then apply your filter to the rows the server sends back, so you increase rather than reduce network traffic when you use a filter in this way. To filter a form or report based on a stored procedure or function, you should include parameters in the procedure and set the form or report Input Parameters property. For more details about input parameters, see Chapter 19 in *Microsoft Office Access 2003 Inside Out*.

Filtering Lookup Values

Combo box controls and list box controls are a good way to look up related values to set a foreign key or to provide a list of potential values for a filter. But keep in mind that for every combo box or list box that uses a table or a query as its row

source, Access must fetch data from the shared data file to fill in the row values. If the table or query returns thousands of rows from a table on the server and the user needs to select only one value, you're asking Access to do more work than necessary. You can deal with this problem in one of two ways: filter the row source or provide an alternative lookup method.

Note: Another alternative is to use a local copy of a lookup table as the row source of a combo box or list box control. See "Pushing the Envelope," page 3xxx, for details.

Restricting Rows in a Combo Box or List Box

The good news is Access optimizes the fetching of the row source for a combo box or list box. Combo boxes are especially efficient because Access doesn't fetch the entire row source until the user begins to type in the box or opens the list. (Access does look up any bound value as you move from row to row.) However, if the row source contains thousands of rows, Access might need to fetch a lot of data over the network as the user searches for the desired value in the list.

Filtering an unbound combo box or list box is relatively easy using a parameter query. For example, the Customer combo box shown in Figure 3-2 has the following query set for its row source:

```
PARAMETERS Forms!fpopOrderSelect!cmbSalesRep Long;
SELECT NullID As CustomerID, "< All Customers >" As CustomerName,
"" As City
FROM tblDummy
UNION ALL SELECT tblCustomers.CustomerID, tblCustomers.CustomerName,
```

```
tblCustomers.City  
FROM tblCustomers  
WHERE (tblCustomers.Active=True) AND  
((tblCustomers.SalesRepID = [Forms]![fpopOrderSelect]![cmbSalesRep]) OR  
([Forms]![fpopOrderSelect]![cmbSalesRep] Is Null))  
ORDER BY CustomerName;
```

Of course, the name of the form is fpopOrderSelect, and the name of the combo box containing the SalesRepID is cmbSalesRep. Code in the form's Load event initially sets the Sales Rep to the ID of the currently signed-on employee and requeries the Customer combo box—which effectively filters the customer list to only those assigned to the current employee. When the user selects a different Sales Rep (the signed-on employee can look at orders entered by other employees), code in the AfterUpdate event of the cmbSalesRep combo box control requeries the Customer combo box control, causing the query shown above to run again and fetch the customers for the newly selected sales rep.

Tip: You might be wondering why the row source of the Customer combo box is a union query. The tblDummy table has exactly one row with a long integer field set to Null. I union this null value with the real list to generate an entry that allows the user to clear the combo box by setting it to null. You might find this technique particularly useful on any search parameter form.

But what about filtering a lookup list for data entry? In a sales and inventory database, you might have thousands of products. If you provide a combo box on the order entry form that allows the user to pick the product to be ordered, the combo box might delay a second or two when the user tries to open the list to select a product to

sell. If your products are categorized, it would make sense to require the user to pick a category first, and then pick a product from a shorter list.

Only 77 products are in the Inventory sample database, but the Orders form does require you to select a category before adding a product to an order. You can see how this works by starting the application (open the frmSplash form), signing on as me, clicking Orders, and searching for all unbilled orders in the month of February 2005. The first order that appears (if you haven't changed the sample data) has ten products ordered. You can go to the new record row in the Products subform, select Dairy Products for Category, and then open the Product list. You should see only the 10 products that are in that category, as shown in Figure 3-3.

Sales & Inventory - Orders

Close Save Cancel Add New Search... Edit All Print Options

Order ID: 1517 Invoiced: ☐

Customer: Ernst Handel

Contact Name: Roland Mendel

Ship To: Kirchgasse 6

City: Graz State/Prov.:

Country: Austria Postal Code: 8010

Order Date: 2/25/2005 Ship Date:

Tax Status: Not taxable Tax Rate: 0.000

Products

Category	Product	Quantity	Cost	Price	P.O.#	Status
Meat/Poultry	Thüringer Rostbratwurst	15	\$79.91	\$123.79	1517007	Allocated
Dairy Products	Geitost	1	\$1.58	\$2.50	0	None
*	Description	Qty. On H:				None
	Geitost	2				
	Gorgonzola Telino	0				
	Gudbrandsdalsost	0				
	Mascarpone Fabioli	0				
	Mozzarella di Giovanni	0				
	Queso Cabrales	4				
	Queso Manchego La Pastora	0				

Record: 14

Figure 3-3.

You must select a category before you can pick a product on the Orders form in the Inventory sample database.

As you might guess, some of the code running behind the subform sets up the combo box to display the restricted list of products. But the subform is a form in Continuous Form view, which presents a special problem when working with filtered combo boxes. Although you can see more than one row in a continuous form, there's actually only one copy of the combo box control. If you filter the combo box to show products from a certain category, the combo box for any other row for a product in a different category will go blank because the product on the other row is no longer in the row source.

To solve this problem, you must include the lookup table in the record source for the form and use a text box that overlays the display portion of the combo box to show the real value from every row. The record source for the fsubOrderProducts form is as follows:

```
SELECT tblOrderProducts.OrderID, tblOrderProducts.LineNo,
tblProducts.Category, tblOrderProducts.ProductID,
tblProducts.Description, tblProducts.SellPrice,
tblOrderProducts.VendorID, tblVendors.VendorName,
tblOrderProducts.Quantity, tblOrderProducts.OrderPONo,
tblOrderProducts.OrderPOLineNo, tblOrderProducts.InvoiceNo,
tblOrderProducts.Cost, tblOrderProducts.Price, tblOrderProducts.Status,
tblOrderProducts.DateAlloc, tblProducts.OrderMultiple AS PROM
FROM tblProducts INNER JOIN
    (tblOrderProducts LEFT JOIN tblVendors
        ON tblOrderProducts.VendorID = tblVendors.VendorID)
    ON tblProducts.ProductID = tblOrderProducts.ProductID;
```

If you open the fsubOrderProducts form in Design view, click on Category and drag down, and then click on Description and drag down, you'll reveal the two overlay text boxes, as shown in Figure 3-4.

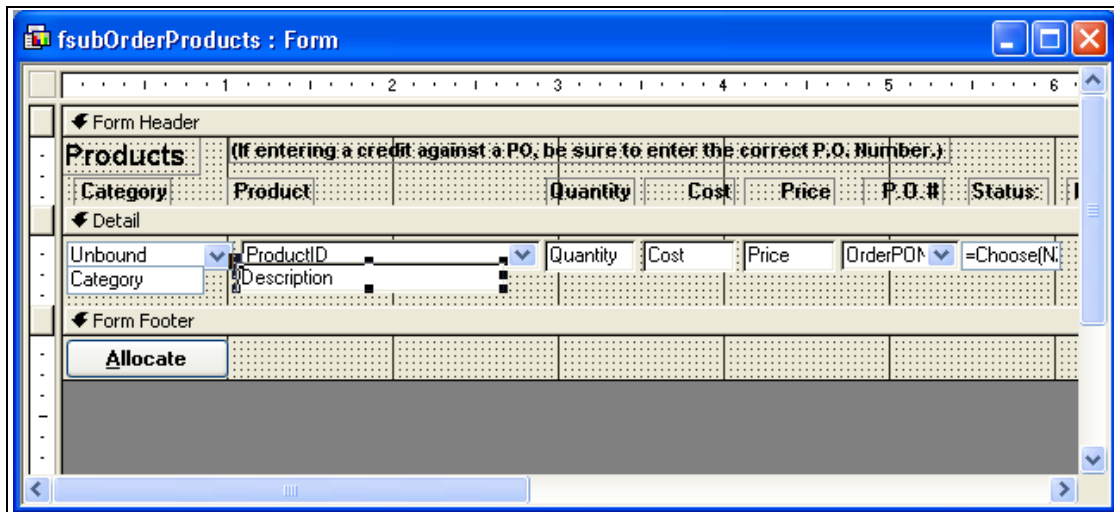


Figure 3-4.

The problem with filtered combo boxes on a continuous form can be solved by using overlaid text boxes.

You can see that the text box sitting on top of the unbound cmbCategory combo box always displays the category from the current row. Likewise, the Description text box sitting atop the cmbProductID combo box displays the product description of the current row (the displayed text in the underlying combo box). The rest of the work happens in code behind the form, which is as follows:

```
Private Sub CategoryName_Enter()  
    ' Move focus to category combo box behind this text box  
    ' when user enters this field  
    Me.cmbCategory.SetFocus  
End Sub  
  
Private Sub cmbCategory_AfterUpdate()
```

```
' Filter the product list based on the new category
Me.cmbProductID.RowSource = "SELECT * FROM qlkpProducts " & _
    "WHERE Category = '" & Me.cmbCategory & "'"
End Sub

Private Sub Description_Enter()
    ' Push focus to combo box behind this control
    Me.cmbProductID.SetFocus
End Sub

Private Sub Form_Current()
    Dim strSQL As String
    ' Sync up the category combo
    Me.cmbCategory = Me.Category
    ' Reset the Row source for the products combo box
    If Not IsNothing(Me.Category) Then
        ' If got a good category, then use it as filter
        Me.cmbProductID.RowSource = "SELECT * FROM qlkpProducts " & _
            "WHERE Category = '" & Me.cmbCategory & "'"
    Else
        ' No category - return an empty set
        ' (will be reset by Category AfterUpdate)
        Me.cmbProductID.RowSource = "SELECT * FROM qlkpProducts " & _
            "WHERE ProductID IS NULL"
    End If
    ' Other code not related to this issue here ...
End Sub
```

The Enter event code for the two overlay text box controls makes sure that should the user click into the control, the focus immediately goes to the combo box control behind it. (The two text box controls are not in the tab order.) The AfterUpdate event code for the cmbCategory combo box resets the row source of the cmbProductID combo box to match the newly selected category. And finally, as the

user moves from row to row, the form Current event code keeps the cmbProductID row source in sync with the category on that row.

Resetting the Row Source property does requery the combo box and forces Access to look up the bound value for the row, but the result is a series of tiny probes to the database instead of a large fetch when the user needs to add a product to an order.

Providing an Alternate Lookup Method

In some cases, you might not be able to logically filter a very large list that you need for a combo box control or list box control. One alternative is to pop open a separate search criteria form for the list every time the user needs to select a new value, but that could become tedious for the user. Another alternative is to provide an unbound text box where the user can begin to type the name they want. You can write code to monitor the keystrokes and do individual quick probes to the database to find the closest matching value and display it. If the user presses Enter or tabs out of the control, your code can set the foreign key value for the name the code looked up as the user was typing.

Although the Membership Tracking sample database has only 35 members defined (using a combo box to look up all member names would be reasonable), the application does contain one form that demonstrates this technique. Open the Membership application (*Membership.mdb*), and open the frmMeetings form. In the sample data, I am not registered yet in the first meeting. Click the Attendees button on the frmMeetings form, and in the Attendee Options dialog box, select the **Open**

the meeting sign-in form option and click Go to open the frmMemberAttendance form. The general purpose of this form is to allow someone sitting at the meeting registration desk to sign in members as they arrive by simply typing in each member's name. When the correct name appears, the user can press Enter or Tab to add the member to the registration list for the meeting and then verify other information on file while the member is still standing there.

In the **Sign In, Please** box, begin typing the first three characters of my last name—**vie**. As soon as you type the third character, the text box fills in my name, as shown in Figure 3-5. (The figure shows the data for the previously signed-in member in the main area of the form.) Press Enter or Tab, and the code behind the form adds my attendance record and displays the information from my record in the tblMembers table so that you can edit it.

Membership - Meeting Sign-In

Meeting Sign-In 14 Sep 2004, 18:00; September 2004 General Meeting

Sign In, Please: Viescas, John (Last, First)

Or edit the member data below:

Title: Mr. **Status:** Active **Default Addr.:** Work

Last Name: Dawson **First Name:** C. Matt **Middle Name:**

Suffix: **Nickname:** Matt **Date Joined:** 5/31/2004

Presenter?: ☐ **Dues Paid?:** ☒

Personal Info **Work Info** **Dues**

Address: 2345 Central Drive

City: Quilcene **Country:** United States

State/Prov.: WA **Phone:** (360) 555-0138

Postal Code: 98376 **Mobile:** (360) 555-0171

Birth Date: 05/26/1972

Email: C. Matt Dawson

Total Signed-In: 16 **Total Dues Paid:** 16 **Print Badges**

Record: 1 of 16 (Filtered)

Figure 3-5.

Using an unbound text box to search for values.

Access signals several events as a user types in a text box. These events include KeyDown, KeyPress, Change, and KeyUp. The KeyDown event occurs when the users presses any key. In this event, you can also detect whether the user is holding down the Shift, Ctrl, or Alt key. When you don't want the key to be delivered to the control—for example, to prevent the user from tabbing out of the control or using Page Up or Page Down in the control—you can set the Keycode to 0 to instruct Access to act as though the user didn't press the key.

The KeyPress event occurs when a character is about to be delivered to the control. This event does not occur, for example, when the user presses one of the arrow keys or the Backspace key. The Change event occurs when a key the user pressed changes the value of the text displayed in the control—for example, the user typed a letter or number or pressed the Backspace key to delete a character. Note that the Value property of the control doesn't change until the user moves out of the control and thereby commits the change to the control. However, you can examine the Text property of the control to find out what's currently displayed. Finally, the KeyUp event occurs when the user releases the key.

All the work to make the lookup happen occurs in the Change and KeyDown events for the text box. The KeyDown event occurs first, so that event procedure gets to look at the Keycode and determine if the user pressed the Backspace or Delete key. If the user is correcting data in the text box using these keys, it doesn't make sense for

the Change event procedure to do anything with these keystrokes—what’s in the Text property of the control might not be what the user needs to look up. Below is the start of the KeyDown event procedure for the sign in text box. The code sets a module variable—intChangeOff—to let the Change event procedure know that it shouldn’t process the next keystroke.

```
Option Compare Database
Option Explicit
Dim lngMemberID As Long, intChangeOff As Integer, strName As String

Private Sub txtSignIn_KeyDown(KeyCode As Integer, Shift As Integer)
Dim strName As String, strWhere As String, intI As Integer
Dim strLast As String, strFirst As String, varID As Variant
Dim rst As DAO.Recordset, lngNewMember As Long
    ' Trap some keys and tell Change to ignore
    If (KeyCode = vbKeyDelete) Or (KeyCode = vbKeyBack) Then
        ' Delete or backspace - ignore
        intChangeOff = True
        ' Reset the saved memberID
        lngMemberID = 0
    Exit Sub
End If
```

When the user types a character that needs to be examined, the KeyDown event leaves the intChangeOff variable set to False. The Change event procedure looks at the current value of the Text up to the position of the cursor (indicated by the SelStart property). As long as the text box contains at least three characters and what the user types changes the leading part of the text, the code attempts to find the matching name. For example, the procedure won’t begin to look up the name until the user types at least the third character of the last name—the letter *e* in my last name.

The code finds me when the user types the letter *e*—it fills in the rest of my name but leaves the cursor sitting between the *e* and the first letter *s*. If the user next types an *s* to replace the letter already there, the code doesn't do another lookup because typing an *s* in the fourth position isn't actually changing what the procedure already found, but the procedure does refill the text box with the found name and positions the cursor again. Below is the beginning of the code in the Change event procedure that performs these tasks.

```
Private Sub txtSignIn_Change()  
    Dim strText As String, intI As Integer, strLast As String, strFirst As String  
    Dim strWhere As String, intStart As Integer  
    Dim db As DAO.Database, rst As DAO.Recordset  
    ' If change is firing because of this code  
    If intChangeOff = True Then  
        ' Bail  
        Exit Sub  
    End If  
    ' Do nothing until the text is at least 3 characters  
    If Len(Me.txtSignIn.Text) < 3 Then  
        Exit Sub  
    End If  
    ' Attempt to do a lookup on the name up to SelStart  
    If Me.txtSignIn.SelStart > 0 Then  
        ' Save the start  
        intStart = Me.txtSignIn.SelStart  
        ' Get the characters up to the cursor  
        strText = Left(Me.txtSignIn.Text, Me.txtSignIn.SelStart)  
        ' If the leading characters didn't change  
        ' (User typed a character that matches what was already displayed)  
        If strText = Left(strName, intStart) Then  
            ' Don't bother to do another lookup
```

```
' Put the name back
Me.txtSignIn = strName
' Make sure intStart not > new length
If intStart > Len(Me.txtSignIn.Text) Then
    intStart = Len(Me.txtSignIn.Text)
End If
' Put the cursor back by setting SelStart
Me.txtSignIn.SelStart = intStart
' Select to end of text
Me.txtSignIn.SelLength = Len(Me.txtSignIn.Text) - intStart
' Done
Exit Sub
End If
```

If the code has not yet found a matching name or the user types a different character in the text box, the Change event parses out the last name and first name by looking for the separating comma. If the user entered a last name, a comma, and a first name, the procedure searches for an exact match on the last name (the text up to the comma) and a wildcard match on the first part of the first name. If the text contains no comma, the procedure performs a wildcard search on the last name only. It then attempts to look up that name in the tblMembers table.

Note: If you want to change this code to search for something like a company name where the name the user types might appear in the middle of the string (the actual name is “The Blue Yonder Company,” but the user typed “Blue”), you can add the wildcard character (*) to both ends of the search string.

If the procedure can’t find the name, it doesn’t change what the user typed, but it does clear the lngMemberID and strName module variables to indicate that the

name typed cannot be found. Here's the code in the Change event procedure that parses the name and performs the lookup:

```
' Find any comma
intI = InStr(strText, ",")
If intI > 0 Then
    strLast = Trim(Left(strText, intI - 1))
    strFirst = Trim(Mid(strText, intI + 1))
    strWhere = "[LastName] = '" & strLast & _
    "' And [FirstName] Like '" & strFirst & "*'"
Else
    strLast = Trim(strText)
    strWhere = "[LastName] Like '" & strLast & "'"
End If

' Point to this database
Set db = DBEngine(0)(0)

' Try to get the member ID and name
Set rst = db.OpenRecordset("Select MemberID, LastName, FirstName " & _
    "FROM tblMembers WHERE " & strWhere & _
    " ORDER BY LastName, FirstName")

' If got one,
If Not rst.EOF Then
    ' Update this textbox
    ' Can't change the Text property because that
    '   fires the stupid Change event again!
    ' Save the new name
    strName = rst!LastName & ", " & rst!FirstName
    Me.txtSignIn = strName
    ' Make sure intStart not > new length
    If intStart > Len(Me.txtSignIn.Text) Then
        intStart = Len(Me.txtSignIn.Text)
    End If
    ' Put SelStart back
    Me.txtSignIn.SelStart = intStart
    ' Select to end of text
```

```
        Me.txtSignIn.SelLength = Len(Me.txtSignIn.Text) - intStart
        ' Update the saved member ID
        lngMemberID = rst!MemberID
    Else
        ' Not found - set clear saved member ID
        lngMemberID = 0
        ' .. and name
        strName = ""
    End If
    ' Close the recordset
    rst.Close
    Set rst = Nothing
    Set db = Nothing
Else
    ' Nothing to search on - clear the saved member ID
    lngMemberID = 0
End If
End Sub
```

When the user presses the Tab key or the Enter key to indicate a desire to accept the displayed value, the KeyDown event gets to look at those keystrokes first. Neither of these keys changes the value of the text, so the code doesn't need to tell the Change event to not process the keystroke. If the user pressed either of these keys, the remaining code in the KeyDown event first checks to see if the Change event code found a matching member ID. If the code in the Change event previously found a match, the code in the KeyDown event attempts to look up that member ID in the form's recordset, which is editing rows in the tblMemberAttend table. If the member ID isn't in the recordset, the member isn't yet signed in to the current meeting. The code adds the member (explained in [Chapter 6, "Tracking Member Activities"](#)), checks for dues paid (explained in [Chapter 7, "Tracking Member Status"](#)), and positions to the new record so the user can verify current address and contact

information. If the attendance record already exists, the code positions the form display to that record.

Finally, if the Change event did not previously find a match on the name, the code in the KeyDown event asks the user if this is a new member. If the user replies Yes, the code sets up a search string to verify that the record has been added and opens the fdlgMemberAdd form and passes that form the text the user entered. If the user successfully adds a new member record with the name originally entered, the code then branches back up to add the member attendance record for that new user. The remaining code in the KeyDown event procedure that performs all these tasks is as follows:

```
' If Tab or Enter,
If (KeyCode = vbKeyTab) Or (KeyCode = vbKeyReturn) Then
    ' Assume they meant to update the control
    ' Set an error trap
    On Error GoTo txtSignIn_Error
UserAdded:
    ' If got a member ID
    If Not IsNothing(lngMemberID) Then
        ' Get a copy of the recordset
        Set rst = Me.RecordsetClone
        ' See if the row exists
        rst.FindFirst "MemberID = " & lngMemberID
        ' If not found
        If rst.NoMatch Then
            ' Save the memberID locally
            lngNewMember = lngMemberID
        ' ** Several lines of code to insert a member
        ' ** attendance record - explained in Chapter 6.
```

```
' ** Several lines of code to check dues ...
' ** Explained in Chapter 7.
Else
    ' Position to the found row
    Me.Bookmark = rst.Bookmark
End If
' Clear the recordset
Set rst = Nothing
Else
    ' Member wasn't previously found by the Change event
    ' See if there is some text
    If Len(Trim(Me.txtSignIn.Text)) > 5 Then
        ' OK, got at least 5 characters - see if they want to add
        strName = Trim(Me.txtSignIn.Text)
        ' Build the verification search string
        ' See if there's a comma
        intI = InStr(strName, ",")
        ' If yes,
        If intI > 0 Then
            ' Parse out the last name
            strLast = Trim(Left(strName, intI - 1))
            ' Parse out the first name
            strFirst = Trim(Mid(strName, intI + 1))
            ' Build the test predicate
            strWhere = "[LastName] = '" & strLast & _
                "' And [FirstName] = '" & strFirst & "'"
        Else
            ' Build a simple last name predicate
            strWhere = "[LastName] = '" & Trim(strName) & "'"
        End If
        ' Verify that they want to add the new Member
        If vbYes = MsgBox("Member " & strName & " is not defined. " & _
            "Do you want to add this Member?", _
            vbYesNo + vbQuestion + vbDefaultButton2, gstrAppTitle) Then
            ' Open the add a Member form and pass it the new value
            DoCmd.OpenForm "fdlgMemberAdd", DataMode:=acFormAdd, _
```

```
        WindowMode:=acDialog, OpenArgs:=strName
    ' Code will wait until "add" form closes
    ' - now verify that it got added!
    varID = DLookup("MemberID", "tblMembers", strWhere)
    ' If not
    If IsNull(varID) Then
        ' Ooops
        MsgBox "You failed to add a Member that matched " & _
            "what you entered. Please try again.", _
            vbInformation, gstrAppTitle
    Else
        ' Save the new ID
        lngMemberID = varID
        ' Loop back up to try again with the newly added row
        GoTo UserAdded
    End If
End If

End If

' Tell the user there's a problem.
MsgBox "The name you entered does not exist!", _
    vbExclamation, gstrAppTitle
End If

' Tell Change to ignore this keystroke
intChangeOff = True
Exit Sub
End If

' Not ignoring key, so turn off ignore flag
intChangeOff = False
txtSignIn_Exit:
Exit Sub

txtSignIn_Error:
    ' Make sure Echo is on
    Application.Echo True
    ' Not supposed to error - tell user
    MsgBox "Unexpected Error: " & Err & ", " & Error
```

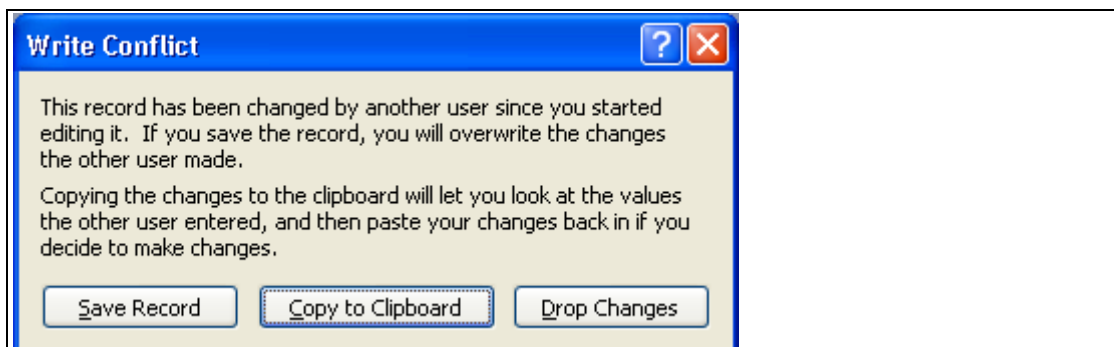
```
' Log it
ErrorLog Me.Name & "_txtSignin_KeyDown", Err, Error
' Bail
Resume txtSignIn_Exit
End Sub
```

Understanding Additional Optimizations

As you have read so far, most of the optimizations you can implement for client/server are centered around minimizing the amount of data you transmit between the client machine and the server when editing data on forms. This section discusses additional points to consider.

Never Open the Same Record for Editing in Multiple Forms

In some applications, one or more subject tables legitimately have dozens of fields. It's very tempting to provide one form to edit some of the most frequently used fields and another to edit other fields. Unless you're careful to make sure that the user can edit in only one form at a time, the user can end up with edits pending for the same record in two different places. Access treats this as though two different users are editing the same record at the same time. If the user has multiple edits pending for the same record, saves one, and then attempts to save the other, Access displays the confusing error message shown in Figure 3-6.



F03YS06.bmp

Figure 3-6.

The warning message Access displays when a user has edited the same record in two different forms at the same time.

Note: Access will also display this message if a user starts to edit a record, another user edits and saves the exact same record, and then the first user attempts to save the record. This is likely to happen only when a table contains a few records, a user starts an edit on a record that some other user also needs to change, waits a long time, and then tries to save. You can trap that error in the form Error event and look for error code 7787. When you encounter that error, you can display your own error message and clear the edits on the form so the user can start over.

The real solution is to use the tab control to allow all fields to be edited on one form. Place the most commonly used fields on the first page of the tab and hide other fields behind other tabs. The user can change the data in fields on one tab and click to another tab to continue editing without problems.

Use a Local Table for Complex Report Data

When a report contains dozens or hundreds of pages and includes calculations that reference group totals (such as a percentage of the total), Access might need to fetch the data for the report from the server multiple times as the user scrolls through the pages and then scrolls back in Print Preview. If the record source of the report is complex and involves three or more tables, the user might also experience noticeable delays when moving between the pages of the report.

You should consider loading the data for this type of report into a local table before opening the report bound to that local table. This working table does not need to follow the design rules for active tables—that is, it can contain repeating groups and data from multiple subjects. You should also consider using this technique when the user needs to print a series of complex reports and all the reports need the same data. Fetch the data from the server once into your local tables, and then open the series of reports.

Avoid More Than Three Subforms on a Form

When your application opens a form that contains multiple subforms, Access must fetch not only the data for the outer form but also the data for each of the subforms, even if the user cannot see all the subforms (perhaps because they're on tab control pages). Even if you are careful to filter the outer form, Access must open multiple threads to the server to be able to fetch all the data. So, for most applications, three subforms is a reasonable limit to avoid performance problems. Although you can nest subforms up to 10 levels deep, you should also avoid nesting more than three levels (a subform on a form that's also the subform on another form). The critical point is that any one form should open no more than four form objects—the outer form and three subforms on that form, or the outer form with one simple subform and a second subform with another form as a subform on that form.

If you have rummaged around in the sample databases, you might have discovered the frmMembers form in the Membership Tracking application (*Membership.mdb*) that has *four* subforms on a tab control (Meetings, Committees,

Dues History, and Officer History), as shown in Figure 3-7. (The other two tabs edit fields in the tblMembers row.) OK, I'm guilty as charged! My justification (or my excuse) is the subforms on the Committees and Officer History tabs are unlikely to have many rows (if any at all) for most members. In truth, if this is a large group with hundreds of rows in each subform for each member, I should find another way to solve the problem.

The screenshot shows a Microsoft Access form titled "Membership - Members". The form has a tab control with six tabs: "Personal Info", "Work Info", "Meetings", "Committees", "Dues History", and "Officer History". The "Personal Info" tab is currently selected. The form contains various input fields for member information, including Title, Last Name, First Name, Middle Name, Suffix, Nickname, Date Joined, Default Address Is, Status, Address, City, State/Prov., Postal Code, Country, Phone, Mobile, Birth Date, and Email. A photo of a man is displayed next to the fields. The form also includes "Add" and "Delete" buttons. The status bar at the bottom shows "Record: 32 of 35".

Figure 3-7.
The frmMembers form in the Membership Tracking application uses four subforms on a tab control.

If you find that a form like this one performs poorly in a client/server application, you can redesign the form to use only one subform control and provide an option group with toggle button controls to simulate the tab control. When the user clicks a button asking for data in a different subform, code that responds to the option group AfterUpdate event can show/hide the appropriate controls and dynamically

load the correct subform into the single subform control. You can find a sample form that demonstrates this technique in the Membership Tracking sample database (*Membership.mdb*). Open the database and then open the frmMembersOneSubform form, as shown in Figure 3-8.

The screenshot shows a Microsoft Access form titled "Membership - Members". The form has a header section with "Members" in a large font and "Member ID: 35" in a text box. Below the header, there are several input fields and controls for member information:

- Title:** A dropdown menu with "Mr." selected. To its right are radio buttons for "Mr.", "Mrs.", "Ms.", and "Dr.", with "Mr." selected.
- Last Name:** Text box containing "Viescas".
- First Name:** Text box containing "John".
- Middle Name:** Text box containing "L.".
- Suffix:** A dropdown menu.
- Nickname:** Text box.
- Date Joined:** Text box containing "5/31/2004".
- Default Address Is:** Radio buttons for "Home" and "Work", with "Home" selected.
- Status:** A dropdown menu with "Honorary" selected.

Below these fields is a row of six buttons: "Personal Info", "Work Info", "Meetings", "Committees", "Dues History", and "Officer History". The "Personal Info" button is currently selected.

The main body of the form displays the following information:

- Address:** 603 W. 13th St. STE 1A PMB 215
- City:** Austin
- State/Prov.:** TX
- Postal Code:** 78701-1796
- Country:** United States
- Phone:** (512) 476-2993
- Mobile:** (empty text box)
- Birth Date:** 09/29/1990 (with a calendar icon)
- Email:** John.Viescas

To the right of this information is a portrait photo of a man with glasses and a mustache. Below the photo are "Add" and "Delete" buttons.

At the bottom of the form, there is a record navigation bar showing "Record: 32 of 35" with various navigation icons.

Figure 3-8.
The frmMembersOneSubform form in the Membership Tracking database that displays data from multiple related tables using only one subform.

This form uses an option group control with toggle button controls to simulate a tab control. You can click each of the buttons to see the other information about each member. When you click the Meetings, Committees, Dues History, or Officer History buttons, the AfterUpdate event of the option group control hides and reveals controls using the Visible property of each and, if necessary, loads the proper form into the Source Object property of the single subform control. The trick is to layer the

controls properly so you don't have to fiddle with too many Visible properties. Figure 3-9 shows you the form in Design view with the layers pulled away from each other.

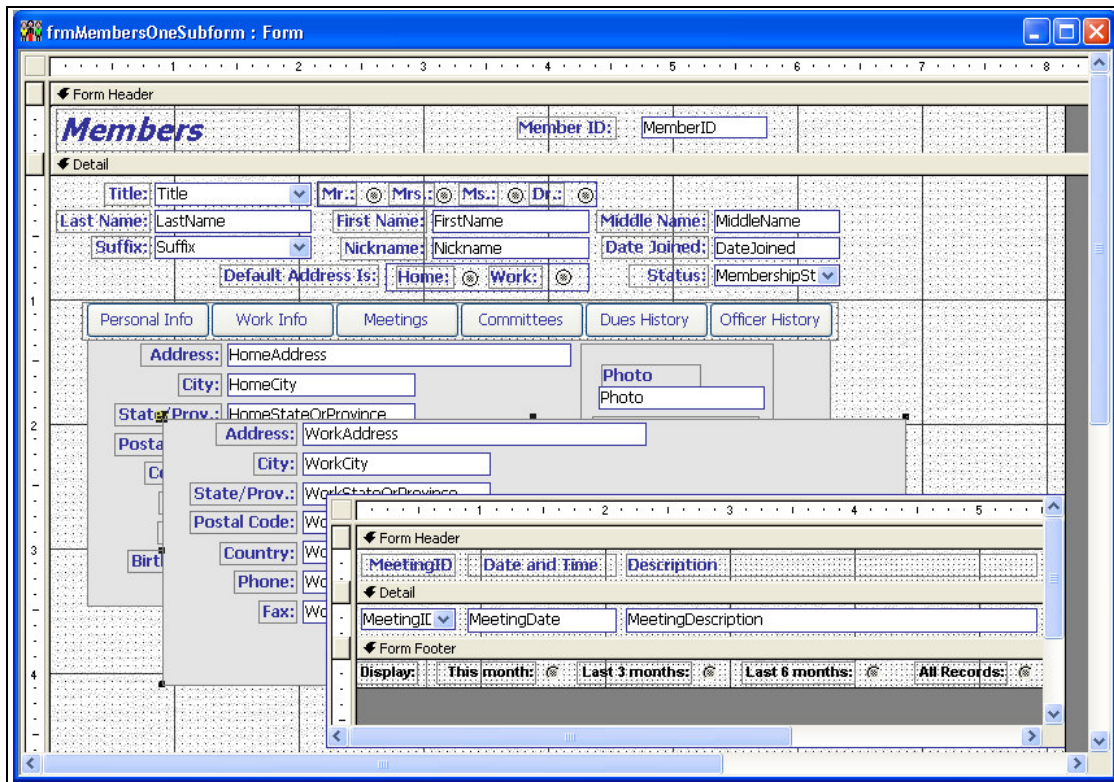


Figure 3-9.
The underlying layers of the frmMembersOneSubform form in the Membership Tracking sample database.

The bottom layer is an image control with its background set to light gray and all the controls for the Personal Info button placed on top. The next layer is another image control (imgPersonalCover) with the controls for the Work Info button on top of that. The final layer is the subform that initially displays the fsubMemberAttend form—the data corresponding to the Meetings button. The subform control is the top layer because you cannot send a subform control behind other controls—it always displays on top. The form also has a tiny text box control—txtSafe—that's not in the tab sequence and is tucked in the corner of the form header. This control provides a

place to park the focus while the code that responds to changes in the option group control works. You can't change any property of a control that has the focus.

Hiding the Personal Info is easy—as long as the `imgPersonalCover` control is visible, it will hide those controls. The code, however, must hide and reveal each of the controls for the Work Info button individually. You can find the code that performs this sleight-of-hand in the `AfterUpdate` event procedure for the option group control. The code is as follows:

```
Private Sub optTabs_AfterUpdate()  
    ' User "clicked" a "tab"  
    ' Hide and reveal stuff  
    ' Put the focus in a safe place  
    Me.txtSafe.SetFocus  
    Select Case Me.optTabs  
        Case 1  
            ' Personal Info  
            ' Hide the subform  
            Me.fsubForm.Visible = False  
            ' Hide all the "work" controls  
            Me.WorkAddress.Visible = False  
            Me.WorkCity.Visible = False  
            Me.WorkCountry.Visible = False  
            Me.WorkExtension.Visible = False  
            Me.WorkFaxNumber.Visible = False  
            Me.WorkPhone.Visible = False  
            Me.WorkPostalCode.Visible = False  
            Me.WorkStateOrProvince.Visible = False  
            ' Unhide the personal info by hiding the  
            ' covering image  
            Me.imgPersonalCover.Visible = False  
            ' Put the focus on the first Personal control
```

```
Me.HomeAddress.SetFocus
```

Case 2

```
' Work Info
' Hide the subform
Me.fsubForm.Visible = False
' Hide the Personal info by unhiding the
' covering image
Me.imgPersonalCover.Visible = True
' Unhide all the "work" controls
Me.WorkAddress.Visible = True
Me.WorkCity.Visible = True
Me.WorkCountry.Visible = True
Me.WorkExtension.Visible = True
Me.WorkFaxNumber.Visible = True
Me.WorkPhone.Visible = True
Me.WorkPostalCode.Visible = True
Me.WorkStateOrProvince.Visible = True
' Put the focus on the first work control
Me.WorkAddress.SetFocus
```

Case 3

```
' Meeting Info
' Load the meeting subform, if necessary
If Me.fsubForm.SourceObject <> "fsubMemberAttend" Then
    Me.fsubForm.SourceObject = "fsubMemberAttend"
End If
' Reveal the subform to hide everything else
Me.fsubForm.Visible = True
' Put the focus there
Me.fsubForm.SetFocus
```

Case 4

```
' Committee Info
' Load the committee subform, if necessary
If Me.fsubForm.SourceObject <> "fsubMemberCommittee" Then
    Me.fsubForm.SourceObject = "fsubMemberCommittee"
End If
' Reveal the subform to hide everything else
```

```
        Me.fsubForm.Visible = True
        ' Put the focus there
        Me.fsubForm.SetFocus
    Case 5
        ' Dues
        ' Load the dues subform, if necessary
        If Me.fsubForm.SourceObject <> "fsubMemberDues" Then
            Me.fsubForm.SourceObject = "fsubMemberDues"
        End If
        ' Reveal the subform to hide everything else
        Me.fsubForm.Visible = True
        ' Put the focus there
        Me.fsubForm.SetFocus
    Case 6
        ' Officer history
        ' Load the officer subform, if necessary
        If Me.fsubForm.SourceObject <> "fsubMemberOfficer" Then
            Me.fsubForm.SourceObject = "fsubMemberOfficer"
        End If
        ' Reveal the subform to hide everything else
        Me.fsubForm.Visible = True
        ' Put the focus there
        Me.fsubForm.SetFocus
    End Select
End Sub
```

Dealing with Complex Subforms: A Case History

For one client, I designed a very complex form that allowed the client to edit all the information for a work order to remanufacture large diesel engines. Each work order could have a log of the work performed, a list of the parts used, the labor charged to the work order, trip charges to go to the client site to perform the repair, services charged at a package price, and charges for work subcontracted out to other

vendors—six subforms in total. I knew before I started that attempting to load data into all seven forms (the outer form and six subforms) might take minutes, not seconds.

My solution was to use only two subform controls. In the first subform, I loaded the parts subform—data required for virtually every work order. When the form opened, the second subform was bound to the labor data—the second most likely to be used set of related data. I placed the two subforms on top of each other and sized them exactly the same. Instead of a tab control, I used an option group control (similar to the frmMembersOneSubform that you can find in the Membership Tracking database) with a set of toggle button controls lined up to look like tabs. When the user clicked Parts or Labor, the code in the option group control AfterUpdate event set the Visible property of the two subforms appropriately. If the user needed to edit any of the other four categories, the code made the second subform visible and loaded the requested subform into it by setting its Source Object property.

Just for fun, I also created a test form with six embedded subforms. Loading a work order that had dozens of rows in each of the related categories took nearly two minutes over a 100BT network. My two subform solution loaded in 5 seconds or less.

Be Careful Where You Place the Linked Data File

If you have many files on the server that manages your desktop database (.mdb) data file, the server probably has a complex folder structure. Unfortunately, Access can take several orders of magnitude longer when it attempts to establish a connection to a file that's buried several folders deep rather than connect to a file

saved in the root folder or in a folder only one level deep. Performance slows significantly when the file is in a folder more than two levels deep and/or the total length of the path name is very long. This is true even when the share that you're using points directly at the folder containing the file. The file server must still maintain locks on the real folder tree, not just the single folder.

One folder that can be particularly misleading is the My Documents folder that exists on every Windows machine. That folder really exists at C:\Documents and Settings\<user name>\My Documents on most machines—three levels deep in the folder structure. If you're having performance problems using linked tables to a desktop database file (.mdb) on a server, try moving the file to the root folder or a folder only one level down.

Validating Links on Startup

When you're using linked tables in a desktop database (.mdb or .mde), you should always include startup code that verifies the links and restores them if necessary. You might need to do this if the user chose a different logical drive name for the share that contains the data file. If the link to the server is down, the application should inform the user and exit gracefully.

All the sample databases include code that runs in the startup form (the frmSplash form in all samples) that performs this task. The core of the procedure is the *CheckConnect* function that you can find in modStartup in all the sample databases. The procedure begins by finding all the attached tables and testing them by

opening a recordset on the table. Note that the code leaves one recordset open during this process to avoid the overhead of rebuilding the connection for each table. This is a technique you can use whenever you are going to open and close several recordsets over a short period of time and no bound form is open. When you leave one recordset open, the subsequent open commands reuse the already established connection. The code is as follows:

```
Public Function CheckConnect() As Integer
' This function called directly by frmCopyright to verify
' connections.
' It is also called by function CheckStartup (called from frmSplash)
' It calls functions AttachAgain and GetFileName as needed
Dim db As DAO.Database, rst As DAO.Recordset, tdf As DAO.TableDef
Dim rstV As DAO.Recordset, intFirst As Integer, intI As Integer
Dim strFilePath As String, strPath As String, varRet As Variant

' Set an Error trap
On Error GoTo CheckConnect_Err
' Point to this database
Set db = DBEngine(0)(0)
' Turn on the progress meter on the status bar
varRet = SysCmd(acSysCmdInitMeter, "Verifying data tables...", _
    db.TableDefs.Count)
' Loop through all TableDefs
For Each tdf In db.TableDefs
' Looking for attached tables
If (tdf.Attributes And dbAttachedTable) Then
' Try to open the table
' Use alternate recordset if already processed the first
If intFirst = True Then
Set rst = tdf.OpenRecordset()
' This one OK - close it
rst.Close
```

```
' And clear the object
Set rst = Nothing
Else
' Doing the first one - use alternate
' recordset and leave it open to speed up the process
Set rstV = tdf.OpenRecordset()
' OK - set "first processed" flag
intFirst = True
End If
End If
' Update the progress counter
intI = intI + 1
varRet = SysCmd(acSysCmdUpdateMeter, intI)
Next tdf
' Got through them all - clear the progress meter
varRet = SysCmd(acSysCmdClearStatus)
' Close the "first" recordset
rstV.Close
' and clear it
Set rstV = Nothing
' Set a good return
CheckConnect = True

CheckConnect_Exit:
Exit Function
```

If any of the open commands fail, the error trap causes execution to resume at the CheckConnect_Err label (shown below). The code resets any error and sets a new error trap to take a different action if the attempt to refresh the links fails. Because I expect that you'll run all the sample databases from your local drive, the code next checks to see if it can find the data file in the same path as the current database. The public constant gstrDataFile contains the expected name of the data file for the application. The code is as follows:

```
CheckConnect_Err:
    ' Clear the error
    Err.Clear
    ' Set new error trap
    On Error GoTo CheckConnect_Err2
    ' If we successfully got the "first" one open
    If intFirst = True Then
        ' Close and clear it
        rstV.Close
        Set rstV = Nothing
    End If
    ' Clear the status bar
    varRet = SysCmd(acSysCmdClearStatus)
    ' First, try to use the current path of this database
    strPath = CurrentProject.Path
    strFilePath = strPath & "\" & gstrDataFile
    ' Use DIR to see if the data file is here
    If Not IsNothing(Dir(strFilePath)) Then
        ' Call the generic re-attach code
        If AttachAgain(strFilePath) = -1 Then
            ' Got a good re-attach
            ' Set OK return
            CheckConnect = True
            ' Done
            Exit Function
        End If
    End If
End If
```

If the code cannot find the file locally, it displays a message telling you to locate the file using the Open File dialog box that is about to appear. When you click OK in the MsgBox dialog box, the code continues and calls the *GetFileName* function that uses the ComDlg class module (not shown) to tell Windows to open the standard Open File dialog box. Unless you click Cancel in that dialog box, the *GetFileName* function returns a new path and file name. If you do not locate a file,

the code issues a warning that you probably won't be able to open any of the linked tables or run the application. The code is as follows:

```
' No success to this point
' Tell the user about the problem
' - about to show an open file dialog box
MsgBox "There's a temporary problem connecting to the " & _
    gstrShortTitle & " data. " & _
    "Please locate the " & gstrShortTitle & _
    " data file in the following dialog box.", _
    vbInformation, gstrAppTitle
' Set up the default file name
strFilePath = gstrDataFile
' Call the file dialog function
If Not (GetFileName(strFilePath)) Then
    ' Tell user of error
    MsgBox "You failed to select the correct file.  WARNING: " & _
        "You may not be able to open any of the linked tables " & _
        "or run the application.  You can open frmCopyright or " & _
        "the startup form (frmStartup) to try again.", _
        vbCritical, gstrAppTitle
    ' Set Failed return
    CheckConnect = False
    ' Done
    Exit Function
End If
```

When you locate a file, the code opens the frmReconnect form that displays a simple message telling you that the attempt to relink the tables is in progress. It calls the *AttachAgain* function one last time with the new file name. If that fails, the code issues a final warning message and exits. The code at the end of the procedure is the second error trap that executes if any of this code generates an error. The code is as follows:

```
' Open the "info" form telling what we're doing
DoCmd.OpenForm "frmReconnect"

' .. and be sure it has the focus
Forms!frmReconnect.SetFocus

' Try calling the attach code again
If AttachAgain(strFilePath) = 0 Then
    MsgBox "Relinking of attached tables failed.  The file " & _
        gstrDataFile & " must " & _
        " be in the folder you located and cannot be renamed." & _
        vbCrLf & vbCrLf & "You can open frmCopyright to try again.", _
        vbCritical, "Building Microsoft Access Applications"

    ' Close and bail
    CheckConnect = False
    Exit Function
End If

' Close the reconnect "splash" form
DoCmd.Close acForm, "frmReconnect"

' All OK
CheckConnect = True
Exit Function

CheckConnect_Err2:
' Got an unexpected error
' Tell user-
MsgBox "Unexpected error checking attached tables. " & Err & _
    ", " & Error, vbCritical
' Log it
ErrorLog "CheckConnect", Err, Error
' Bail
CheckConnect = False
' Close the info form if it is open
If IsFormLoaded("frmReconnect") Then DoCmd.Close acForm, "frmReconnect"
' ... bail
Resume CheckConnect_Exit
End Function
```

In both the case where the code finds the file in the local folder or when you specify a path and file via the Windows Open File dialog box, the code calls the *AttachAgain* function to reattach the linked tables. Similar to the code in the *CheckConnect* function, this procedure loops through all the tables in the TableDefs collection looking for attached tables. When it finds one, it updates the Connect property of the TableDef object to point to the new location and uses the RefreshLink method to try to validate the new location. If Access cannot successfully perform the RefreshLink, it generates an error that is handled by the code at the Err_Attach label. The procedure might have successfully relinked some of the tables at the point of the failure, but when any RefreshLink fails, the procedure doesn't try any other tables and exits to the calling procedure with a False (failure) return. If the code manages to loop through all the tables successfully, it returns True to the calling procedure to indicate that the entire relink was successful. The code is as follows:

```
Public Function AttachAgain(strFilePath As String) As Integer
' This is a generic function that accepts a new path name
' and attempts to refresh the links of all attached tables
' Input: File Path name as C:\SomeFolder\SomeSubFolder\SomeDB.mdb
' Output: True if successful
Dim db As DAO.Database, tdf As DAO.TableDef, rst As DAO.Recordset
Dim varRet As Variant, intFirst As Integer
Dim intI As Integer, intK As Integer, intL As Integer
' Get a pointer to the database
Set db = DBEngine(0)(0)
' Set the "first table" indicator
intFirst = True
' Turn on the progress meter
varRet = SysCmd(acSysCmdInitMeter, "Reconnecting Data...", _
    db.TableDefs.Count)
```

```
' Set an error trap
On Error GoTo Err_Attach

' Attempt to reattach the tables
intI = 0 ' Reset the status meter counter
For Each tdf In db.TableDefs
    ' Looking for attached tables
    If (tdf.Attributes And dbAttachedTable) Then
        ' Change the Connect property to point to the new file
        tdf.Connect = ";DATABASE=" & strFilePath
        ' Attempt to refresh the link definition
        tdf.RefreshLink
        ' If the first table, then open a recordset
        ' to make this go faster
        If (intFirst = True) Then
            Set rst = db.OpenRecordset(tdf.Name)
            intFirst = False
        End If
    End If
Next tdf

' Update the status counter
intI = intI + 1
' .. and update the progress meter
varRet = SysCmd(acSysCmdUpdateMeter, intI)
' And pause for a sec so the status bar updates
DoEvents

' Done - clear the progress meter
varRet = SysCmd(acSysCmdClearStatus)
' Clear the object variables
Set tdf = Nothing
rst.Close
Set rst = Nothing
Set db = Nothing

' Return attach successful
AttachAgain = True
Attach_Exit:
Exit Function
```

```
Err_Attach:
    ' Uh, oh - failed.  Write a log record
    ErrorLog "AttachAgain " & strFilePath, Err, Error
    ' Clear the progress meter
    varRet = SysCmd(acSysCmdClearStatus)
    ' Clear the object variables
    Set tdf = Nothing
    Set db = Nothing
    ' Return attach failed
    AttachAgain = False
    ' Exit
    Resume Attach_Exit
End Function
```

Pushing the Envelope

You can use the techniques that you've learned to this point in this chapter to successfully build a client/server application with desktop database files (.mdb) that can support up to 20 simultaneous users. These techniques also work best when you need to share less than 200 megabytes of data. If you need to support more users or need to store more data, you can take one of two paths: upsize your application to SQL Server, or radically restructure how your desktop application works.

Most developers choose to upsize. But if your company (or your client) doesn't want to move up to the cost of SQL Server (both the cost of the software and the extra expertise needed to manage SQL Server on an ongoing basis), you can "push the envelope" with a desktop database architecture.

The sample Customer Support application demonstrates one way to design a client/server application that can support more simultaneous users with an Access

application that uses a desktop database file (.mdb) to store the data. If you open the application database file (**Support.mdb**) and look at the list of tables, as shown in Figure 3-10, you'll see what looks like local copies of some of the linked tables stored within the database.

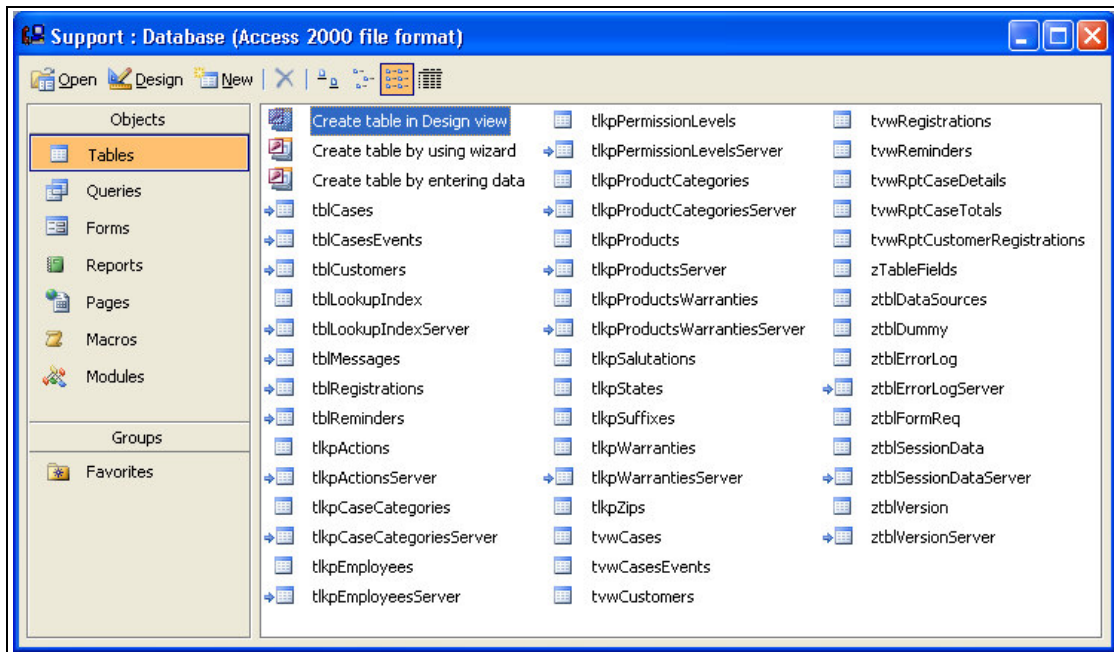


Figure 3-10.
The list of tables in the Customer Support application.

For example, you can find customer support case data in both the `tblCases` table and the `tvwCases` table. The data in the `tlkpEmployees` table appears to be the same as the data in the `tlkpEmployeesServer` table. Here's what's happening:

- For tables that the application updates frequently (such as those for customers, support cases, support case events, product registrations, and reminders), the application fetches from the server only the data that the user needs to edit. The application uses a combination of bound and unbound forms to present the data and to allow the user to edit it. When a form is bound to data, it uses a copy of the data that the application has fetched into a local table. (The local tables containing

a subset of data from the server have a name beginning with “tvw” (short for table view). To save the data, the application collects the changes from the unbound form and writes those changes back to the copy on the server.

- For tables that won’t change very often (such as those for employees, case action codes, products, product categories, and so on), the application keeps an exact copy of the data stored on the server. The application periodically checks the status of records on the server that might have been updated by other users and refreshes the local copies only when necessary. In the Customer Support sample application, these tables have names beginning with “tlkp” (lookup tables).

Taking this approach isn’t for the faint of heart. If you look behind the forms and into the code modules in the Customer Support database, you’ll find lots of code that is necessary to manage the movement of data from and to the shared data file on the server. As you’ll learn later in **Part 4**, “Implementing a Customer Support Application,” the actual tasks implemented in the application are quite simple.

Editing Data Locally

What allows the Customer Support application to handle 50 or more users is the fact that the application severely limits the need to work with data in the data file on the server. Unlike a classic client/server application using linked tables, the application is designed to never edit data directly from the server. Only a few of the forms, combo boxes, and list boxes are directly bound to data, and the bound data all resides in a local table containing a copy of the data.

Examining an Unbound Form

You can actually open several of the forms directly from the Database window. For example, you can open the frmCustomers form that the application uses to display and edit customer data, and it will look like Figure 3-11.

The screenshot shows a Microsoft Access form titled "Customers" with a subtitle "Customer Registration". The form is organized into several sections. The top section contains fields for "Salutation", "First Name", "Middle", "Last Name", and "Suffix". Below this is a "Default Address" field. The form is then split into two main columns. The left column is for "Home Address" and includes fields for "Address 1", "Address 2", "City", "State / Prov", "Postal", "Country", and "Phone". The right column is for "Business Address" and includes fields for "Company Name", "Address 1", "Address 2", "City", "State / Prov", "Postal", "Country", and "Phone". Below these columns are fields for "Cell Phone", "Fax", and "Email". At the bottom of the form, there are four fields: "Product", "Serial #", "Date of Purchase", and "Warranty". The form has a blue title bar and a green footer bar with the name "John Viescas" and a small icon. There are also three buttons at the bottom: "New Case", "View Cases", and "New Product Registration".

Figure 3-11.

The frmCustomers form shows no data when you open it from the Database window.

Notice that the form is completely blank. However, the form does display customer information when you open it from the application. The application forces the user to choose from a local list of customer records that the application fetches from the server in response to search criteria. The user can pick the one record that's needed, and code in the application opens the frmCustomers form and tells the form which record to fetch.

In Chapter 16, “Organizing Customer Information,” you’ll learn how the application manages customer information. To see the frmCustomers form with a specific customer displayed for editing, be sure the frmCustomers form is not open, press Ctrl+G to open the Immediate window, and enter the following command:

```
DoCmd.OpenForm "frmCustomers", OpenArgs:="65"
```

The argument passed to the form tells it to locate and load the information about the customer whose customer ID is 65. When you go back to the Access window, you should see the form with the data displayed for the requested customer, as shown in Figure 3-12.

Customers

Customer Registration

Salutation	First Name	Middle	Last Name	Suffix
Mr.	Michael Greg		Blythe	

Default Address: Business

Home Address		
Address 1		
1234 2nd Boulevard		
Address 2		
City	State / Prov	Postal
Pacific House	CA	95726
Country	Phone	
United States	(530) 555-0104	

Business Address		
Company Name		
Fabrikam		
Address 1		
1234 3rd Boulevard		
Address 2		
City	State / Prov	Postal
Pollock Pines	CA	95726
Country	Phone	
United States	(530) 555-0144	

Cell Phone	Fax	Email
(530) 555-0176	(530) 555-0145	MBlythe@NoMail.Please

	Product	Serial #	Date of Purchase	Warranty
▶	DW100 Dishwasher	1590562344	5/12/2005	90 Days Full Warranty
▶	M50 Micro Oven	1818417370	5/3/2005	90 Days Full Warranty
▶	RF850 Primo Refrigerator	1092540145	5/16/2005	3 Year Full Warranty

New Case View Cases New Product Registration

John Viescas

Figure 3-12.

The frmCustomers form displaying data for customer number 65.

Note: The application doesn't ever reveal the customer ID value to the user because the actual number assigned to each customer is irrelevant. The user always searches for customers by name, phone number, or e-mail address. However, the application internally uses the customer ID to efficiently identify the correct record.

The only bound data in the form is the product list in the subform window. If you change any of the data, code behind the form reveals an informational message in the blank area next to the Default Address label telling you that you need to click the Save Customer button (the button with the disk and pencil icon) to save your changes. You can click any of the arrows in the list of products in the subform window, and the application opens the frmRegistrations form to display the details about the customer's registered product.

Loading Data to Edit

Each form that allows the user to edit unbound data must fetch the requested record and set the controls on the form to display the data. This is in stark contrast to a normal bound form where the form fetches the records and displays the data in bound controls. Each form contains a *fctReLoadRecord* function that is specific to the data the user can edit in the form, but the operation of this function is very similar across all forms.

The *fctReLoadRecord* function depends on the value of a hidden control to either indicate that the user is editing a new record (the control contains 0 or Null), or specify the unique identifier of the record that should be loaded. In the frmCustomers

form, the txtCustomerID text box control serves this purpose. You can open the frmCustomers form in Design view and find the txtCustomerID text box with its Visible property set to No just to the left of the new record command button. When the form opens, the form's Open event procedure saves the customer ID value that was passed in the OpenArgs argument in the txtCustomerID control and calls the *fctReloadRecord* function to fetch the data to display.

The beginning of the function checks to see if the txtCustomerID control contains nothing (is 0 or Null). When the ID is nothing, the code clears all the editable controls by setting them to Null. In some forms, the code in this section sets the equivalent of a default value (for example, the current date and time) in some controls. Basically, the code is simulating what Access does when the users chooses to move to a new record in a bound form. The code is as follows:

```
Function fctReloadRecord() As Boolean
' Called when the form opens, the user asks to start a new
' record, or the user cancels an edit after a save failed.
' Function either clears all editable controls or reloads
' the customer specified by the current CustomerID.
On Error GoTo errfctReloadRecord
' If CustomerID is empty,
If IsNothing(Me.txtCustomerID) Then
' .. clear the form to create an empty record
Me.cmbSalutation = Null
Me.txtCFirst = Null
Me.txtCMiddle = Null
Me.txtCLast = Null
Me.cmbCSuffix = Null
Me.txtHomeAddress1 = Null
Me.txtHomeAddress2 = Null
```

```
Me.txtHomeCity = Null
Me.cmbHomeStateOrProvince = Null
Me.txtHomePostal = Null
Me.txtHomeCountry = Null
Me.txtHomePhone = Null
Me.txtBusinessName = Null
Me.txtBusinessAddress1 = Null
Me.txtBusinessAddress2 = Null
Me.txtBusinessCity = Null
Me.cmbBusinessStateOrProvince = Null
Me.txtBusinessPostal = Null
Me.txtBusinessCountry = Null
Me.txtBusinessPhone = Null
Me.txtCellPhone = Null
Me.txtFax = Null
Me.txtEmail = Null
```

When the txtCustomerID text box control contains a value, the code calls another public function—*LoadRecord*—that is used by all forms to load data from a requested table into the form. (You can find the *LoadRecord* function in the modRecordEditor module.) In this case, the code calls that function, passing parameters that identify the name of the calling form, the name of the table containing the data to load, the identifier of the record requested, and a title to use in error messages. If the record load fails, the code exits.

Next, the code calls the *fctRefreshReg* function (that you can find in the module for the frmCustomers form) that clears the tvwRegistrations table and uses an insert query to load the product registration data for the current customer into that table. If you look at the design for the fsubCustomersRegistrations form that is the subform on the frmCustomers form, you'll find that it is bound to a query on the local

tvwRegistrations table. (The code for the *fctRefreshReg* function is described later in this chapter.) Finally, the code marks the record saved, hides the label that informs the user when the record has been changed, requeries the subform to display the new data in the tvwRegistrations table, and exits. The code is as follows:

```
Else
    ' Otherwise, load the record using the CustomerID
    ' Call the function that loads the record
    If LoadRecord("frmCustomers", "tblCustomers", _
        Me.txtCustomerID, "Customer") = False Then
        ' Got an error - bail
        Exit Function
    End If
    ' Load the related registrations into the view table
    If fctRefreshReg(Me.txtCustomerID) = False Then
        ' Got an error - bail
        Exit Function
    End If
End If
'After loading the new record,
' set the Boolean 'saved' markers to true (or saved).
blnSaved = True
' .. and hide the save instruction label
Me.lblSaveCustomer.Visible = False
'Refresh the sub form display
Me.fsubCustomersRegistrations.Requery
' Return success
fctReLoadRecord = True
okfctReLoadRecord:
Exit Function
errfctReLoadRecord:
' Got an error - set failure
fctReLoadRecord = False
' Display and log the error
```

```
ErrTryingTo "load the customer data"  
Resume okfctReLoadRecord  
End Function
```

The *LoadRecord* function is one of several functions you'll find in the modRecordEditor module that help manage unbound data. Most of these functions use a local table that defines how to map fields in each table to controls on the forms that edit them. The table also identifies which fields must have a value (Required), which fields must always be saved (isSaved), and which field is the primary key (isID). Table 3.1 shows you a sample of some of the rows that control how the code works with customer data.

Table 3-1. Sample Data from the ztblFormReq Table

FilterName	FieldName	ControlName	Description	Required	isID	isSaved
frmCustomers	CFirst	txtCFirst	First Name	Yes	No	Yes
frmCustomers	CLast	txtCLast	Last Name	Yes	No	Yes
frmCustomers	CMiddle	txtCMiddle	Middle Initial	No	No	Yes
frmCustomers	CustomerID	txtCustomerID	Customer ID	No	Yes	No

Note: All the tables have a primary key defined using the AutoNumber data type to simplify acquiring a unique key when adding new records. This avoids an additional probe to the server to determine the previous high value when inserting a row. As a consequence, the ID fields in the tables are not marked required or saved in the ztblFormReq table.

As you saw earlier, the *fctReLoadRecord* function in the frmCustomers form calls the *LoadRecord* function in the modRecordEditor module to fetch the data for a

specific customer ID. The *LoadRecord* function uses information from the ztblFormReq table to know which fields to fetch from the tblCustomers table and which controls should be assigned each field value. The code begins by verifying that the target form named in the strFormName parameter is open. The code next fetches the rows from the ztblFormReq table for the specified form. The code then searches within the rows for the one that indicates the primary key of the table—the isID field is True—and saves the name of the key field. Note that if any of these operations fail, the code raises an error, which causes the error code at the end of the function to display and log the error and exit. The code is as follows:

```
Public Function LoadRecord(strFormName As String, strTableName As String, _
    lngRecordID As Long, Optional strRecordDesc As String = "form") _
    As Boolean
'*****
'LoadRecord - loads a record as requested by any open form
'Inputs:
'    strFormName - The name of the form making the request
'    strTableName - The name of the table to load the record from
'    lngRecordID - The identifier for the record to be loaded
'    strRecordDesc (optional) - A description of the record to load
'Output:
'    Returns a Boolean value to indicate if the record was loaded or not
'*****
Dim db As DAO.DataBase
Dim rstF As DAO.Recordset
Dim rst As DAO.Recordset
Dim frm As Form
Dim strSQL As String
Dim strIDName As String
    ' Set an error trap
    On Error GoTo errLoadRecord
```

```
' Make sure the form specified is open
If IsFormLoaded(strFormName) = False Then
    ' Form should be there - raise an error
    Err.Raise vbObjectError + 1024, , _
        "Invalid call to Load Record function."
Else
    ' Get a pointer to the form
    Set frm = Forms(strFormName)
End If

' Point to this database
Set db = DBEngine(0)(0)

' Get the rows that tell us what to load
strSQL = "SELECT FR.* FROM ztblFormReq AS FR " & _
    "WHERE FR.FilterName = '" & strFormName & "'"
Set rstF = db.OpenRecordset(strSQL)

' Make sure there are some
If rstF.EOF Then
    ' Close the recordset
    rstF.Close
    ' Raise an error
    Err.Raise vbObjectError + 1025, , _
        "No field rows found for requested form."
End If

' Find out the primary key of the table to update
rstF.FindFirst "isID = True"

' If not found
If rstF.NoMatch Then
    ' Close the recordset
    rstF.Close
    ' Raise an error
    Err.Raise vbObjectError + 1026, , _
        "Key field not found for record load."
End If

' Save the name of the key field
strIDName = rstF!FieldName

' Go back to start of the recordset
```

```
rstF.MoveFirst
```

Now that the code knows the name of the primary key field, it opens the one record needed from the data file on the server to fill the controls in the form. The code then loops through all the rows for the form in the ztblFormReq table, using the control name and field name information to assign the values from the record to the controls in the form. After processing all fields, the code closes all recordsets and returns the value of the primary key to the calling procedure. The code is as follows:

```
' Get the record to load
strSQL = "SELECT TBL.* FROM " & strTableName & " AS TBL " & _
    "WHERE TBL." & strIDName & " = " & lngRecordID
Set rst = db.OpenRecordset(strSQL, dbOpenDynaset, dbSeeChanges)
' If not found,
If rst.EOF Then
    ' Close the recordsets
    rstF.Close
    rst.Close
    ' Raise an error
    Err.Raise vbObjectError + 1027, , _
        "Record requested to load not found."
End If
' Loop through all the fields
Do While Not rstF.EOF
    ' Set the form control value from the field
    frm(rstF.Fields("ControlName")) = _
        rst.Fields(rstF.Fields("FieldName"))
    ' Move to next row
    rstF.MoveNext
' Loop until done
Loop
' Close out
rst.Close
```

```
rstF.Close
LoadRecord = True
okLoadRecord:
    ' Clean up
    Set rstF = Nothing
    Set rst = Nothing
    Set db = Nothing
    Set frm = Nothing
    Exit Function
errLoadRecord:
    ' Trapped an error - set failed return
    LoadRecord = False
    ' Display and log the error
    ErrTryingTo "load the " & strRecordDesc & " record"
    Resume okLoadRecord
End Function
```

This code is specifically designed to make a single connection to the data in the file on the server, quickly fetch the single record needed, grab the field values, and then disconnect. As a result, the application code running on each workstation is using the network and the shared data file for only a few milliseconds. The user might work on the fetched record for several minutes before needing to connect to the shared data file again to save any changes.

In this design, the workstation code maintains no permanent connection to the server. A classic client/server application that uses linked tables has a connection as long as any form is open. Keep in mind that Access refreshes any open recordset based on the Refresh Interval setting on the Advanced tab in the Options dialog box (choose Options from the Tools menu). You can certainly change this setting, but most developers leave the Refresh Interval set to the default 60 seconds.

In some cases, 60 seconds is too long. But choosing a lower number in a standard client/server application can severely impact performance. If the user opens a form and then goes off to lunch, that form's recordset is being refreshed by Access once a minute. Constant polling contributes to limiting the typical client/server application to no more than 20 users.

Validating and Saving Unbound Data

One of the challenges of using unbound data is that your forms won't automatically trap data type, data length, and validation rule violations as the user enters data into the form. You must write code to validate the data as the user enters it. You can certainly trap errors when you try to write the data to the shared data file, but that might be too late for the user to fix the problem easily.

The Customer Service application implements rudimentary data validation on each form. Wherever possible, the form controls have either a Format property or Input Mask property or both defined. Even when a control is unbound, Access won't let the user type data that doesn't match the Format or Input Mask property. For example, you can enter only a value recognizable as a date in a control that has its Format property set to Short Date. If you try to type something that Access doesn't recognize as a date/time value, Access displays an error message.

In addition to taking advantage of the Format and Input Mask properties, code in the AfterUpdate event of all the controls that can contain text calls a validation function to test that the user hasn't entered text longer than the field in the table accepts. You can find several examples in the frmCustomers form. For example, code

in the AfterUpdate event procedure for the txtCellPhone text box control makes sure that the user doesn't enter more than 20 characters. The code also calls the fctDirty function in the form module to mark the record changed. The code is as follows:

```
Private Sub txtCellPhone_AfterUpdate()  
    ' Call the length validator  
    Call CheckFieldSize(Me.txtCellPhone, "Cell Phone", 20)  
    ' Mark the record changed  
    fctDirty  
End Sub
```

The application uses the *CheckFieldSize* function that you can find in the modRecordEditor module to verify the field length for controls on many of the forms. The calling procedure passes the control to be verified, a descriptor to use in an error message, and the maximum length allowed. The function tests the length of the data in the current value of the control, shortens it to the maximum length if it is too large, and warns the user that the data has been truncated. The code is as follows:

```
Public Function CheckFieldSize(ctl As Control, strFieldName As String, _  
    intMaxSize As Integer)  
    ' This function checks the length of a value entered in a control.  
    ' Inputs: Control object to validate  
    '           Name of the field for warning message  
    '           Maximum length  
    ' Output: Data truncated to max length, if necessary  
    Dim strValue As String  
    Dim strMsg As String  
    ' Set an error trap  
    On Error GoTo errCheckFieldSize  
    ' Get the value of the control  
    strValue = Nz(ctl.Value, "")  
    ' See if user typed too much
```

```
If Len(strValue) > intMaxSize Then
    ' Warn the user -
    strMsg = "The " & strFieldName & _
        " field can only accept a maximum of " & _
        intMaxSize & " characters. " & _
        "The field value has been truncated to the maximum size."
    Call CustomError(strMsg)
    ' .. and truncate the value
    ctl.Value = Left(ctl.Value, intMaxSize)
End If

okCheckFieldSize:
    Exit Function

errCheckFieldSize:
    ErrTryingTo "validate field " & strFieldName
    Resume okCheckFieldSize

End Function
```

Each form that allows the user to edit unbound data also has a function, *fctCheckSaved*, that verifies and saves any changes. This function runs whenever the user clicks the Save Record button, asks to begin a new record after changing an existing one, or clicks the Close button to close the form.

The function first checks to see if the ID field on the form (the txtCustomerID text box in the case of the frmCustomers form) has a value. If it has no value, the code assumes that the user is saving a new record. If it has a value, the code in some forms next creates any calculated values that should be saved in the record. In the case of the frmCustomers form, the code creates the calculated CustomerName value by concatenating the first and last names with an intervening blank. Next, the code checks the blnSaved module variable to determine if any value has been changed by

the user. If no changes need to be saved, the code returns a confirmation that the data is saved and exits. The code is as follows:

```
Public Function fctCheckSaved() As String
    'This code checks that the required information is complete and then
    'saves the record and flags the linked table for an update
    Dim strErrMsg As String
    Dim blnNewRec As Boolean
    Dim lngID As Long
    ' Set an error trap
    On Error GoTo errfctCheckSaved
    ' If no customer ID yet,
    If IsNothing(Me.txtCustomerID) Then
        ' Set the new record flag
        blnNewRec = True
    Else
        ' Have a customer ID - save it
        lngID = Me.txtCustomerID
        blnNewRec = False
    End If
    'Make sure the composite Customer Name field is properly filled
    Me.txtCustomerName = Me.txtCFirst & " " & Me.txtCLast
    'If the current record already saved,
    If blnSaved = True Then
        ' Confirm saved and exit
        fctCheckSaved = "Saved"
```

When the user has changed any data on the form, the code calls the *ValidateRecord* function that you can find in the modRecordEditor module, passing to the function the name of the form and the value of the filter to be used to verify the data. Remember from Table 3-1 that the ztblFormReq table has a Required field that indicates which fields must have values in the record to be saved. As you'll see later in this section, the *ValidateRecord* function returns an empty string if it finds no

errors and a string containing the fields that failed the validation if any required fields do not have data. When the *ValidateRecord* function returns an error string, the *fctCheckSaved* function displays the error and exits. The code is as follows:

```
Else
    ' Validate the customer data - returns ZLS if successful
    strErrMsg = ValidateRecord("frmCustomers", "frmCustomers")
    ' If the error string not empty,
    If strErrMsg <> "" Then
        ' Set up the message to display
        strErrMsg = "The record could not be saved " & _
            "because the following fields are not complete: " & _
            vbCrLf & strErrMsg & vbCrLf & "What would you like to do?"
        ' Display the message - if user clicked cancel,
        If CustomError(strErrMsg, OkCancel, "Could not save record", _
            Question) = "Cancel Changes" Then
            ' Return that edit was canceled
            fctCheckSaved = "Cancel"
            ' Clear all data to a new screen
            Call fctReLoadRecord
        Else
            ' User clicked - OK, wants to work on it some more
            ' Return that edit failed, but still dirty
            fctCheckSaved = "Ok"      'The user chose to keep working.
            ' Make sure record not marked saved
            fctDirty
        End If
    End If
```

When all required fields have a value, the code calls the *SaveRecord* function that you can find in the *modRecordEditor* module, passing it the name of the form, the filter value for the *ztblFormReq* table, the name of the table in the shared data file on the server to update, the ID of the record to update (0 if adding a new record), and a descriptive string for error messages. (The missing parameter you see in the code

below is an optional local table name where the function saves a copy of the updated record.) The *SaveRecord* function returns the nonzero value of the ID of the record if the save is successful. When the function returns a successful save value, the code sets the *blnSaved* module variable and hides the informative message on the form telling the user to click the Save button. The code is as follows:

```
Else
    ' No error messages, so save the record
    ' *** Additional code here explained in Chapter 16 ***
    'This might take a while...
    DoCmd.Hourglass True
    'If there was no error message,
    ' and the record hasn't been saved then:
    ' Save the Customer - returns the CustomerID if successful
    lngID = SaveRecord("frmCustomers", "frmCustomers", _
        "tblCustomers", lngID, , "Customer")
    ' If customer save successful.
    If lngID <> 0 Then
        ' Save the Customer ID in the form control
        Me.txtCustomerID = lngID
        ' Set up return success
        fctCheckSaved = "Saved"
        ' Turn on the saved flag
        blnSaved = True      'The current record is saved
        ' Hide the "click to save" message
        Me.lblSaveCustomer.Visible = False
    End If
End If

End If

'Done!
okfctCheckSaved:
    ' Make sure hourglass is off
    DoCmd.Hourglass False
```

```
Exit Function
errfctCheckSaved:
    ' Display and log error
    ErrTryingTo "save customer data"
    ' Clear the record so we don't get an unending loop
    fctReLoadRecord
    Me.txtCustomerID = 0
    Resume okfctCheckSaved
End Function
```

The *ValidateRecord* function is quite simple. After verifying that the calling form is open, the code fetches the rows from the *ztblFormReq* table that have the Required field set to True for the specified table filter. The code processes each row, testing that the related form control has a value. When it finds any missing values, it creates an error string message that lists the fields that must have values, using the descriptive field from the *ztblFormReq* table. When the code has finished processing all rows, it returns the contents of the error string—but the string is empty if there were no missing required fields. The code is as follows:

```
Public Function ValidateRecord(formname As String, _
    FilterName As String) As String
    '*****
    'ValidateRecord - Validates a record using the required fields list in
    ' the ztblFormReq table.
    'Inputs:
    ' FormName - The name of the form making the request
    'Output:
    ' Returns a string containing the failed validation information
    ' -- empty if no validation failed.
    '*****
    Dim db As DAO.DataBase
    Dim rst As DAO.Recordset
```

```
Dim frm As Form
Dim strSQL As String
Dim strErrMsg As String

' Set an error trap
On Error GoTo errValidateRecord

' Make sure the form specified is open
If IsFormLoaded(formname) = False Then
    ' Form should be there - raise an error
    Err.Raise vbObjectError + 99, , _
        "Invalid call to Validate Record function."
Else
    ' Get a pointer to the form
    Set frm = Forms(formname)
End If

' Point to this database
Set db = DBEngine(0)(0)

' Set up to find the required fields for the current form
strSQL = "SELECT FR.* FROM ztblFormReq AS FR " & _
    "WHERE FR.Required = True AND FR.FilterName = '" & _
    FilterName & "'"

' Get the list of required fields
Set rst = db.OpenRecordset(strSQL)

' Process all the rows
Do While Not rst.EOF
    ' Use the name of the control from the recordset
    '   to test that the form control contains a value
    If IsNothing(frm(rst!ControlName)) Then
        ' Value is missing - add it to the error message
        strErrMsg = strErrMsg & " -" & _
            rst.Fields("Description") & vbCrLf
    End If
    ' Process the next one
    rst.MoveNext
Loop

' Close the recordset
rst.Close
```

```
' Return the error string
ValidateRecord = strErrMsg
okValidateRecord:
    Set rst = Nothing
    Set db = Nothing
    Set frm = Nothing
    Exit Function
errValidateRecord:
    ' Return an error string
    ValidateRecord = "Error"
    ' Display and log the error
    strErrMsg = "The following error occurred while " & _
        "trying to verify the form information: " & _
        vbCrLf & "Error Number: " & Err.Number & vbCrLf & _
        "Error Description: " & Err.Description
    Call CustomError(strErrMsg, , "Error verifying form information", Severe)
    ' Bail
    Resume okValidateRecord
End Function
```

Finally, let's take a look at the *SaveRecord* function in the *modRecordEditor* module. As you might expect, the code is the reverse of the *LoadRecord* function—this function writes data to the tables instead of fetching data from the tables. The code also handles inserting new records, and it can optionally write the same data to both the master copy in the shared data file on the server as well as to any local mirror copy in the application database.

The code begins by verifying that the specified calling form is loaded. It then fetches the rows from the *ztblFormReq* table for the specified table filter. These rows indicate either fields that must be saved or the ID field of the table. The code searches the recordset for the ID field and saves the field name. If any of these operations fail,

the code raises an error to call the error handling code at the end of the function.

Finally, the code either saves any nonzero record ID or sets a new record flag. The

code is as follows:

```
Public Function SaveRecord(strFormName As String, _
    strFilterName As String, strTableName As String, _
    Optional lngRecordID As Long, _
    Optional strLocalTableName As String, _
    Optional strRecordDesc As String = "form") As Long
'*****

'SaveRecord - saves a record as requested by any open form
'Inputs:
'    strFormName - The name of the form making the request
'    strTableName - The name of the table to save the record to
'    lngRecordID (optional) - The identifier for the record to be saved.
'        If a lngRecordID is provided, then the save is an edit;
'        otherwise a new record is created.
'    strLocalTableName (optional) - The name of the local table
'        that also must be updated.
'    strRecordDesc (optional) - A description of the record to save
'Output:
'    Returns a long integer value containing the
'        saved record ID (0 if the save failed or was cancelled)
'*****

Dim db As DAO.DataBase
Dim rstF As DAO.Recordset
Dim rst As DAO.Recordset
Dim frm As Form
Dim strSQL As String
Dim strIDName As String
Dim intI As Integer
Dim blnNew As Boolean
Dim blnTrans As Boolean

' Set an error trap
```

```
On Error GoTo errSaveRecord

' Make sure the form specified is open
If IsFormLoaded(strFormName) = False Then
    ' Form should be there - raise an error
    Err.Raise vbObjectError + 4096, , _
        "Invalid call to Save Record function."
Else
    ' Get a pointer to the form
    Set frm = Forms(strFormName)
End If

' Point to this database
Set db = DBEngine(0)(0)

' Get the list of fields to update
strSQL = "SELECT FR.* FROM ztblFormReq AS FR " & _
    "WHERE (FR.isSaved = True " & _
    "Or FR.isID = True) " & _
    "AND FR.FilterName = '" & strFilterName & "'"
' .. and open a recordset on it
Set rstF = db.OpenRecordset(strSQL)

' Make sure there are some
If rstF.EOF Then
    ' Close the recordset
    rstF.Close
    ' Raise an error
    Err.Raise vbObjectError + 4097, , _
        "No field records found for requested table."
End If

' Get the name of the ID field
rstF.FindFirst "isID = True"
' If not found
If rstF.NoMatch Then
    ' Close the recordset
    rstF.Close
    ' Raise an error
    Err.Raise vbObjectError + 4098, , _
        "ID field not found for record save."
```

```
End If

' Save the name of the ID field
strIDName = rst!FieldName

' If passed a record ID
If Not lngRecordID = 0 Then
    ' Set the new record flag to false
    blnNew = False
Else
    ' No ID - set the new record flag
    blnNew = True
End If
```

Next, the code starts a transaction to ensure that both the linked data file and the local copy are updated or neither is updated if any error is encountered. The code starts a loop to first process the linked table and then the local table on the second pass. On the first pass, the code determines if a new record is needed and opens a recordset to either insert a new row or fetch the existing row filtered by the ID value specified. When the code is in the second pass through the loop, it checks to see if the optional local table name was supplied and, if so, prepares a new record to insert or fetches the local table record. When updating an existing record, the code raises an error if the record cannot be found in either the linked table or the local table. The code is as follows:

```
' Start a transaction to ensure both updates work
BeginTrans
blnTrans = True

' Loop twice to process server on the first pass
' and optional local copy on the second pass
For intI = 1 To 2
    ' If we're on the first pass,
    If intI = 1 Then
```

```
' And adding a record,
If blnNew = True Then
    ' Open the server table for insert only
    Set rst = db.OpenRecordset(strTableName, dbOpenDynaset, _
        dbAppendOnly)
    ' .. and start a new record
    rst.AddNew
Else
    ' Fetch the record to update from the server
    strSQL = "SELECT " & strTableName & ".* FROM " & strTableName & _
        " WHERE " & strTableName & "." & strIDName & " = " & lngRecordID
    Set rst = db.OpenRecordset(strSQL, dbOpenDynaset, dbSeeChanges)
    ' Make sure we got it
    If rst.EOF Then
        ' Close the recordset
        rst.Close
        ' Raise an error
        Err.Raise vbObjectError + 4099, , _
            "Record requested to update on server not found."
    End If
    ' Put it in Edit mode
    rst.Edit
End If
Else
    ' Second pass - see if there's a local table to update
    If strLocalTableName <> "" Then
        ' Yes - and if new record,
        If blnNew = True Then
            ' Open the local table for insert only
            Set rst = db.OpenRecordset(strLocalTableName, dbOpenDynaset, _
                dbAppendOnly)
            ' .. and start a new record
            rst.AddNew
            ' Copy the lngRecordID set on the server
            rst(strIDName) = lngRecordID
        Else
```

```
' Not a new record - fetch the record to update
strSQL = "SELECT " & strLocalTableName & ".* FROM " & _
    strLocalTableName & _
    " WHERE " & strLocalTableName & "." & _
    strIDName & " = " & lngRecordID
Set rst = db.OpenRecordset(strSQL)
' Make sure we got it
If rst.EOF Then
    ' Close the recordset
    rst.Close
    ' Raise an error
    Err.Raise vbObjectError + 4100, , _
        "Local record requested to update not found."
End If
' Put it in Edit mode
rst.Edit
End If
Else
    ' No local table - Done
    Exit For
End If
End If
```

The remainder of the code within the two-pass loop processes the rows from the `ztblFormReq` table and uses the information in the rows to fetch the value in a control on the form and assign it to the correct field in the recordset being updated. Just before saving the changed record, the code saves the ID of the record. Because all the tables use an AutoNumber data type as the ID field, the JET database engine sets the ID of any new record as soon as any field value is assigned. After executing the two loops to process the linked and local tables, the code commits the transaction to make all changes permanent and returns the ID of the record saved. The code is as follows:

```
' Go to the first field record
rstF.MoveFirst

' Loop and process all the fields
Do While Not rstF.EOF

    ' Update only the ones where isSaved = True
    If rstF!isSaved Then

        ' Copy the form control value to the field
        rst(rstF!FieldName) = _
            frm(rstF!ControlName)

    End If

    ' Process the next field
    rstF.MoveNext

' Loop until done
Loop

' Save the record ID
lngRecordID = rst(strIDName)

' Save the record
rst.Update

' Close the recordset
rst.Close

' Go process the possible second one
Next intI

' Good to here - commit the transaction
CommitTrans

blnTrans = False

' Close the fields recordset
rstF.Close

' Return the ID of the record saved
SaveRecord = lngRecordID
okSaveRecord:

' Clean up
Set rstF = Nothing
Set rst = Nothing
Set db = Nothing
Set frm = Nothing
Exit Function
```

```
errSaveRecord:
    ' Set an error return
    SaveRecord = 0
    ' If a transaction started
    If blnTrans = True Then
        ' Roll it back
        blnTrans = False
        Rollback
    End If
    ' Display and log the error
    ErrTryingTo "save the " & strRecordDesc & " record"
    Resume okSaveRecord
End Function
```

Again, the application is connecting to the shared data on the server only for the few milliseconds required to fetch, update, and save the record. No permanent connection to the shared data file exists after the update completes.

Loading Related Data into a Local Table for Viewing

For many tasks in a database application, you as the developer need to present the user with a list of items to view or edit. Obvious examples include list boxes and combo boxes that provide lookup values. Another common example is a subform that displays a list of rows from a table related to the single row displayed in the outer form—for example, the order details for an order, the contact events for a support case, or the products owned by a customer.

In Figure 3-12 on page 3xxx, you can see a list of products owned by the customer being edited in the outer form. Because the design of this application seeks to avoid binding any forms or controls to the shared data on the server, the application must load the data into a local table. If you look at the design of the

frmCustomersRegistrations form in the Customer Support sample database

(*Support.mdb*), you'll find that the form is bound to the local tvwRegistrations table, not the linked tblRegistrations table. In fact, if you open the tvwRegistrations table in Datasheet view, you'll find records for only one customer ID, not the entire set of records in the tblRegistrations table.

Code in the frmCustomers form loads selected data into the local tvwRegistrations table when the application opens the form to edit a specific customer. In "Loading Data to Edit" on page 3xxx, you learned that the *fctReloadData* function in the module for the frmCustomers form calls the *fctRefreshReg* function to load the data used by the subform. The code deletes all previous rows in the tvwRegistrations table and then executes an insert query to fetch the rows from the tblRegistrations table on the server for the current customer. The code is as follows:

```
Function fctRefreshReg(lngID As Long) As Boolean
' Called from the fctReloadRecord function
' to reload the local view of the registrations for
' the current customer. Input: CustomerID
Dim db As DAO.Database
' Set an error trap
On Error GoTo ErrRefresh 'Enable error trapping
' Point to this database
Set db = DBEngine(0)(0)
' Empty the local Call Events table (delete everything).
db.Execute "DELETE * FROM tvwRegistrations", dbFailOnError
' Update the table with the registrations
' from the source table on the server
db.Execute "INSERT INTO tvwRegistrations " & _
```

```
"SELECT tblRegistrations.* FROM tblRegistrations " & _  
"WHERE tblRegistrations.CustomerID = " & lngID  
'Refresh the view of the registrations on the form  
Me.fsubCustomersRegistrations.Requery  
' Clear the database object  
Set db = Nothing  
' Return successful  
fctRefreshReg = True  
RefreshOk:  
Exit Function  
ErrRefresh:  
' Error - set failure  
fctRefreshReg = False  
' Display and log the error  
ErrTryingTo "reload customer product registrations"  
' Bail  
Resume RefreshOk  
End Function
```

Again, the application connects to the server for a few milliseconds to retrieve the rows needed to display on the form. Once the form is open, the application has no permanent connection to the server, so it avoids tying up valuable network resources.

Using and Refreshing Local Lookup Tables

In addition to local tables that the application loads with data to edit as needed, the application also maintains several tables that are mirror images of the data in the shared file on the server. These tables have the following characteristics:

- The data is not changed frequently.
- The data is needed often to fill lookup lists—combo boxes, list boxes, and some subforms.

- The tables are much smaller than the active tables that are updated frequently.

The tables with these characteristics in the Customer Support application include:

- The list of lookup tables (tblLookupIndex)
- The list of final actions for customer support cases (tlkpActions)
- The list of employees who can sign on to the application (tlkpEmployees)
- The list of issue categories for support case events (tlkpCaseCategories)
- The list of permissions granted to each employee (tlkpPermissionLevels)
- The list of product categories (tlkpProductCategories)
- The list of products (tlkpProducts)
- The list of warranties available for each product (tlkpProductsWarranties)
- The list of warranties offered (tlkpWarranties)

Because the application frequently needs to fetch and display the entire list of available rows from these tables, it doesn't make sense to go to the server to refresh the list every time the application opens a form containing a list box or combo box that needs the data. However, other users might update the data, so the application needs a mechanism to periodically check for updates and reload the local copies.

When the Customer Support application starts up, it runs the *fctLookupUpdate* function that you can find in the modStartup module in the sample **Support.mdb**

database. This function uses information in the `tblLookupIndex` and `tblLookupIndexServer` tables to determine which tables need to be refreshed. These tables contain one row per lookup table, and each row lists the name of the local table, the name of the server linked table, and an update index counter for each update made to the table, as shown in Table 3-2.

Table 3-2. Sample Data from the `tblLookupIndex` Table

TableName	ServerTableName	UpdateIndex
<code>tblLookupIndex</code>	<code>tblLookupIndexServer</code>	1
<code>tlkpActions</code>	<code>tlkpActionsServer</code>	1
<code>tlkpEmployees</code>	<code>tlkpEmployeesServer</code>	1
<code>tlkpCaseCategories</code>	<code>tlkpCaseCategoriesServer</code>	1
<code>tlkpPermissionLevels</code>	<code>tlkpPermissionLevelsServer</code>	1
<code>tlkpProductCategories</code>	<code>tlkpProductCatetoriesServer</code>	1
<code>tlkpProducts</code>	<code>tlkpProductsServer</code>	1
<code>tlkpProductsWarranties</code>	<code>tlkpProductsWarrantiesServer</code>	1
<code>tlkpWarranties</code>	<code>tlkpWarrantiesServer</code>	1

Each time the application writes a change to any of the tables, it increments the update index in both the local copy and the copy on the server. When another copy of the application starts on another machine, the *`fctLookupUpdate`* function compares its local `tblLookupIndex` table with the copy on the server to quickly

determine which tables need updating. The code in the function does this by opening a recordset on a query that joins the two lookup update tables and compares the UpdateIndex fields. For each record returned where the update index doesn't match, the code deletes the old data in the local copy and executes an insert query to refresh the data from the copy on the table. The code also updates its local UpdateIndex field to match the value on the server. The code is as follows:

```
Public Function fctLookupUpdate()  
Dim dbs As DAO.Database, rst As DAO.Recordset, strSQL As String  
    ' Reload the "lookup" tables - all the tables  
    '   whose names begin with "tlkp" as well as the  
    '   tblLookupIndex table.  
    ' Function called periodically to refresh the local  
    ' copies of these tables with any updates others  
    ' have made to the data on the server.  
    ' Set an error trap  
On Error GoTo errfctLookupUpdate  
    ' Point to this database  
Set dbs = DBEngine(0)(0)  
    ' Build query to find updated tables -  
    '   The table tblLookupIndexServer on the server  
    '   contains the latest "lookup index" for each  
    '   lookup table. Query finds tables with a higher  
    '   index than the local copy.  
strSQL = "SELECT LI.TableName, LI.ServerTableName, LI.UpdateIndex " & _  
    "FROM tblLookupIndexServer AS LI " & _  
    "INNER JOIN tblLookupIndex " & _  
    "ON LI.LookUpID = tblLookupIndex.LookUpID " & _  
    "WHERE LI.UpdateIndex > [tblLookupIndex].[UpdateIndex] "  
    ' Open a recordset on the query  
Set rst = dbs.OpenRecordset(strSQL)  
    ' If we have some records  
Do While Not rst.EOF
```

```
' Clear the local copy
strSQL = "DELETE " & rst!TableName & ". * FROM " & rst!TableName
dbs.Execute strSQL, dbFailOnError

' Fetch the latest rows from the server
strSQL = "INSERT INTO " & rst!TableName & _
    " SELECT " & rst!ServerTableName & _
    ". * FROM " & rst!ServerTableName
dbs.Execute strSQL, dbFailOnError

' Update the local update index counter
strSQL = "UPDATE tblLookupIndex " & _
    "SET tblLookupIndex.UpdateIndex = " & rst!UpdateIndex & _
    " WHERE tblLookupIndex.TableName = '" & rst!TableName & "'"
dbs.Execute strSQL, dbFailOnError

' Process the next
rst.MoveNext

Loop

' Close out
rst.Close

Set rst = Nothing
Set dbs = Nothing

okfctLookupUpdate:
Exit Function

errfctLookupUpdate:
' Log the error and tell the user
ErrTryingTo "update lookups"

' Bail

Resume okfctLookupUpdate

End Function
```

Of course, checking for updates when the application starts makes perfect sense. But what if a user leaves the application running for days at a time? If you look at the design of the frmCSMain form that is the main switchboard for the application, you'll find a Timer event procedure. This procedure runs every four minutes, and

every third time (once every 12 minutes), the code calls the *fctLookupUpdate* function to refresh the local tables.

To see how the application manages the updates to these lookup tables, you can open the frmDBAdminProducts form, as shown in Figure 3-13. This form is designed in Continuous Form view to display all the available products. The user can filter the list by setting filter values in the form's footer section. The user must click one of the selector buttons (the right arrow next to each row) to copy the data into the unbound area in the form's header section to be able to make changes.

Category:	Product Name:
Cooking	C200 Cooktop
Cooking	C350 Cooktop
Cooking	C1500 Cooktop
Cooking	C1550 Primo Cooktop
Cooking	O311 Oven
Cooking	O500 Oven
Cooking	O550 Primo Oven
Cooking	M50 Micro Oven
Cooking	M130 Microwave
Cooking	M322 Microwave

Figure 3-13.
Using a form to edit data in a system lookup table.

Although this form could allow the user to edit the data directly from the local lookup table, it's more reliable to use unbound data because any updates must be written to both the local copy and the linked copy on the server. Like all the other unbound editing forms, this form has an *fctCheckSaved* function to validate and save the data. After calling the *ValidateRecord* function to check for data in required fields, the code calls the *SaveRecord* function to save the record. However, note that the code provides both the local table name and the linked table name to update.

Finally, the code calls the *FlagLookup* function that you can find in the modRecordEditor module to update the lookup index values for the table it just changed. Note that the code protects the calls to the two data update functions with a transaction to ensure that either all changes are saved successfully or none are. The code is as follows:

```
Private Function fctCheckSaved()  
    ' This code checks that the required information  
    ' is complete and then saves the record and  
    ' flags the linked table for an update  
    Dim strMsg As String, blnTrans As Boolean  
    ' If already saved,  
    If blnSaved = True Then  
        ' Return success  
        fctCheckSaved = "Saved"  
        ' and exit  
        Exit Function  
    End If  
    ' Set an error trap  
    On Error GoTo errfctCheckSaved  
    ' Call record validation to check for required data  
    strMsg = ValidateRecord("frmDBAdminProducts", "frmDBAdminProducts")  
    ' Return message is zero length string (ZLS) if OK  
    If strMsg <> "" Then  
        ' Build the error message  
        strMsg = "The record could not be saved because " & _  
            "the following fields must be completed: " & _  
            vbCrLf & vbCrLf & strMsg & vbCrLf & _  
            "What would you like to do?"  
        ' Ask the user what to do  
        If CustomError(strMsg, OkCancel, "Save Product Record", _  
            Question) = "Cancel Changes" Then  
            ' User gives up - return cancel
```

```
fctCheckSaved = "Cancel"
Else
    ' User wants to continue edit - return OK
    fctCheckSaved = "Ok"
End If
' Make sure still marked not saved
blnSaved = False
Else
    ' No error - Start a transaction to protect all updates
    BeginTrans
    blnTrans = True
    ' Call the record saver to
    ' update both the remote table
    ' and the local copy.
    If SaveRecord("frmDBAdminProducts", "frmDBAdminProducts", _
        "tlkpProductsServer", Me.txtProductID, _
        "tlkpProducts", "Product") <> 0 Then
        ' Got a nonzero answer - OK
        ' Return success
        fctCheckSaved = "Saved"
        ' Mark the record saved
        blnSaved = True
        ' Flag the lookup table that we have changed it!
        If FlagLookup("tlkpProductsServer", True) = True Then
            ' Both changes done - commit all changes
            CommitTrans
            blnTrans = False
            ' Rebuild my recordset to show any added record
            Me.Requery
        Else
            ' Failed to flag - rollback!
            Rollback
            blnTrans = False
            ' Return failure
            fctCheckSaved = "OK"
            ' Leave record marked not saved
```

```
        blnSaved = False
    End If
Else
    ' Something failed -
    Rollback
    blnTrans = False
    fctCheckSaved = "Ok"
    ' Leave record marked not saved
    blnSaved = False
End If
End If
'Done!
okfctCheckSaved:
    ' Make sure hourglass is off
    DoCmd.Hourglass False
    Exit Function
errfctCheckSaved:
    ' If a transaction in process,
    If blnTrans = True Then
        ' Clear it
        blnTrans = False
        ' and roll it back
        Rollback
    End If
    ' Display and log error
    ErrTryingTo "save customer data"
    ' Clear the record so we don't get an unending loop
    Me.txtProductID = 0
    fctReloadRecord
    Resume okfctCheckSaved
End Function
```

The *FlagLookup* function that you can find in the modRecordEditor module is very simple. The code first fetches the record from the tblLookupIndexServer table for the lookup table that was just updated. It adds 1 to the value of the UpdateIndex

field and writes the record back. If the caller asked to update the local lookup index table as well, the code uses an update query to change the UpdateIndex field to match the value it just saved in the linked table to the server data. The code is as follows:

```
Public Function FlagLookup(strSvrTblName As String, _
    Optional blnSetLocal As Boolean) As Boolean
    '*****
    ' FlagLookup - Updates the value for a specific
    ' table record in the lookup table. This tells all
    ' other user copies to update their local table
    ' versions the next time the value is checked.
    'Inputs:
    '   strSvrTblName - The name of the server
    '                   table to increment by 1
    '   blnSetLocal - Indicates whether the local table
    '                 should also be incremented
    '   (False if the local version is already up to date)
    '*****
    Dim db As DAO.DataBase
    Dim rst As DAO.Recordset
    Dim strSQL As String
    Dim intIndex As Integer
    Dim blnTrans As Boolean
    ' Set an error trap
    On Error GoTo errFlagLookup
    ' Point to this database
    Set db = DBEngine(0)(0)
    ' Start a transaction to protect both updates
    BeginTrans
    blnTrans = True
    ' Fetch the index row for the specified table on the server
    strSQL = "SELECT tblLookupIndexServer.UpdateIndex " & _
        "FROM tblLookupIndexServer " & _
        "WHERE tblLookupIndexServer.ServerTableName = '" & strSvrTblName & "'"
    Dim rst As DAO.Recordset
    Set rst = db.OpenRecordset(strSQL, db.OpenSnapshot, db.LockTypes.NoLock)
```

```
Set rst = db.OpenRecordset(strSQL, dbOpenDynaset, dbSeeChanges)

' Edit the record to lock it
rst.Edit

' Get the current index value
intIndex = rst!UpdateIndex

' Add 1
intIndex = intIndex + 1

' Change the value in the record
rst!UpdateIndex = intIndex

' Save it
rst.Update

' Close the recordset
rst.Close

' If we need to update the local copy, too...
If blnSetLocal = True Then
    ' Use an UPDATE query
    strSQL = "UPDATE tblLookupIndex " & _
        "SET UpdateIndex = " & intIndex & _
        " WHERE tblLookupIndex.ServerTableName = '" & strSvrTblName & "'"
    db.Execute strSQL, dbFailOnError
End If

' Done - commit the updates
CommitTrans

blnTrans = False

' Return success
FlagLookup = True

okFlagLookup:
Set rst = Nothing
Set db = Nothing
Exit Sub

errFlagLookup:
' Got an error - display and log it
ErrTryingTo "flag the " & strSvrTblName & " lookup"
Resume okFlagLookup

End Sub
```

In **Part 4**, “Implementing a Customer Support Application,” you’ll learn all the details about how the application is designed to track customers, capture support issues and events, track follow-up contacts, and report statistics on issue trends. However, it should be clear at this point that the Customer Support application also teaches you how to minimize contact with the shared data stored on a server in order to maximize the number of users your application can support.

The Bottom Line: Keep the Wire “Cool”

If you walk away with nothing else in this chapter, you should understand that minimizing the amount of data transmitted over the network—keeping the wire “cool,” not “hot”—is the key to a successful client/server application. You cannot create a classic single-user application and then just put it on the server for multiple users to share. You can probably split a simple single-user application into a “code” database file (containing queries, forms, reports, and modules) with linked tables to tables in a “data” database file on a file server and successfully deploy the application for three to five users. When you need to support more users, you might need to completely rewrite or revamp your application design.

So, if you know in advance that your application must eventually support multiple users, design how the application code works with client/server in mind from the beginning. When you remember that the main purpose of a database is to share data, nearly every application you write will someday “grow up” to be shared by more than one user. So, even if you think you’re building a simple single-user

application, you'll be better off using good client/server coding techniques from the start.