Chapter 20
Advantage and Visual FoxPro

Note: This chapter accompanies the book Advantage Database Server: A Developer's Guide, 2nd Edition, by Cary Jensen and Loy Anderson (2010, ISBN: 1453769978). For information on this book and on purchasing this book in various formats (print, e-book, etc), visit: http://www.JensenDataSystems.com/ADSBook10

We want to thank Chris Franz from the Advantage team at Sybase. The project discussed in this chapter is based on a Visual FoxPro project originally written by Chris. His design replicates the sample projects created for Delphi and Visual C# presented in Chapters 17 and 18 of this book.

Visual FoxPro is the most recent development environment to benefit directly from Advantage support. Specifically, while Advantage has supported FoxPro CDX files since its earliest days, Advantage now supports Visual FoxPro style tables. This means that Visual FoxPro developers can continue to use the same file format as before, but now do so in a transaction-supporting, remote database environment.

While using Visual FoxPro tables through Advantage provides you with stability and performance that was previously unavailable, most new development in Visual FoxPro, development that does not need to support legacy applications, will want to use the Advantage ADT file format. In addition to supporting the many field types to which you are accustomed, the ADT format provides additional field types, including the new Unicode fields. In addition, using ADT tables in conjunction with Advantage data dictionaries provides you with levels of security and features not supported by the Visual FoxPro format.

This chapter provides you with examples of using the Advantage from Visual FoxPro to perform a wide range of common data-related tasks. As is the case with the other chapters in Part III, this discussion assumes that you are already familiar with the Visual FoxPro development environment. Instead, the focus is on code that works with the data dictionary you have been using throughout this book.

Advantage and Visual FoxPro

If you are using Visual FoxPro tables in a local file server style, without the use of Advantage, you continue using these tables in the same way that you have in the past. The only difference is that is may be possible to use these same tables from other applications using the Advantage Local Server (ALS).

If you want to use these same tables using the Advantage Database Server (ADS), you will need to use one of Visual FoxPro's remote data mechanisms. In particular, Visual FoxPro provides three mechanisms for accessing your data remotely. These are remote views, SQL pass-through, and cursor adapters.

All three of these data access mechanisms make use of industry standard drivers for their data access. Remote views and SQL pass-through make use of ODBC (open database connectivity) drivers, and cursor adapters make use of OLE DB providers. When accessing your Advantage data using these mechanisms you will make use of the Advantage ODBC driver or the Advantage OLE DB provider.

As a result, before you can access your Advantage data using one of these three data access mechanisms, you will have to install either the Advantage ODBC driver or the Advantage OLE DB Provider (or both). In addition, you will also have to provide for the installation of the appropriate driver on any machine to which you wish to deploy your Advantage-enabled Visual FoxPro applications.

When using cursor adapters, installing the Advantage OLE DB provider is sufficient. Using OLE DB Providers, your connection string takes responsibility for configuring how your Advantage driver will perform.

By comparison, when using either remote views or SQL pass-through, installing the ODBC driver is often not enough. You will also have to configure an ODBC data source name (DSN). You will then refer to this DSN when you connect to the Advantage ODBC driver. As must be obvious, a DSN must be configured on any machine on which you will deploy your Advantage-enabled Visual FoxPro applications.

SQL pass-through is the most flexible mechanism in Visual FoxPro for accessing remote data, and as a result, the example application described in this chapter makes use of it. As a result, in order to use this application, you must first install the Advantage ODBC driver. In addition, you need to define a data source name named DemoADD.

You have a choice of a number of different types of DSNs when configuring an ODBC driver. You can create a user DSN, a system DSN, or a file DSN. System DSNs have the advantage that they can be used by any application on a particular computer, regardless of which user is logged in when the program is run. This means that system DSNs can be used by any user. User DSNs, by comparison, are specific to a particular user, and can only be accessed when that user is logged onto the machine. File DSNs provide a middle level of access, in that they can be shared by more than one user, but only if the corresponding ODBC driver has been installed for access by those users.

If you are unfamiliar with configuring an ODBC driver, please see Chapter 23: *ADS with ODBC, PHP, and DBI/Perl*. Note, however, that in most cases, you will want to create either a user DSN or a system DSN, with the system DSN being preferable. There are complications associated with file DSNs. As a result, if you want to create and use a file DSN for your connection, please refer to the Advantage Knowledgebase, which provides insight into using file DSNs with Visual FoxPro.

Note: Chapter 23 describes how to define a DSN for the Advantage ODBC driver. It also describes how to use the ODBC driver using an ODBC connection string. When using the Advantage ODBC driver with Visual FoxPro, you must define a DSN. Configuring the Advantage ODBC driver through a connection string is not an option.

Performing Basic Tasks with Advantage and Visual FoxPro

This section describes some of the more common tasks that you can perform with Visual FoxPro. These include creating a connection, querying a table, using a parameterized query, and executing a stored procedure.

Creating the Connecting

You connect to a data dictionary or a directory in which free tables are located using the SQLCONNECT command. In the project associated with this chapter, this command is executed from the LOAD event of the project's form, named frmSPTDemo, as shown in the following code sample:

```
* declare some public variables for the form
* a connection handle
PUBLIC m.nConn as Integer
* SQL Statement for the runquery procedure
PUBLIC m.lcSQLStatement as String
* connect to an Advantage ODBC connection
m.nConn = SQLCONNECT("DemoADD", "adsuser", "password")
```

This code begins by creating two public variables that will be used repeatedly in these examples. The first variable, m.nConn, is an integer used to hold the connection handle created by the SQLCONNECT function. The second public variable, m.lcSQLStatement, is a string variable used to hold a query definition that is executed by a public method named RUNQUERY.

RUNQUERY is a method associated with the project's form. RUNQUERY is described in the following section.

The actual connection is created through the call to the SQLCONNECT function. In this example, SQLCONNECT is passed three parameters. The first parameter is the name of a configured ODBC DSN, which is DemoADD in this example. If you want to run this code example, you will first have to create a DSN named DemoADD.

The second and third parameters are the user name that you want to use to connect to the data dictionary and that user's password. Please also note that this user, adsuser in this case, must have sufficient rights to the various objects in the data dictionary. If you have worked through all of the examples in the first 16 chapters of this book, adsuser should have those rights.

Just as the LOAD event is used to create a connection, and its associated handle, the form also includes an UNLOAD event, which is used to close the connection. The following is the code associated with the UNLOAD event:

```
* disconnect all connections
SQLDISCONNECT(m.nConn)
```

Executing a SELECT Query

You retrieve data from a table or view by executing a SELECT query. As mentioned in the previous section, a method, named RUNQUERY was specifically created in order to run simple SQL statements, specifically those that do not require parameters. This method executes the query assigned to the m.lcSQLStatement public string variable. The following is the code associated with the RUNQUERY method:

```
lcErrMsg = "Error"
m.nResult = SQLEXEC(m.nConn, m.lcSQLStatement, "curResults")
IF m.nResult > 0
  thisform.grResult.RecordSource = "curResults"
  RETURN
ENDIF
IF m.nResult = 0
  this.grResult.RecordSource = ""
  RETURN
ELSE
  * display error
  AERROR(laErr) && Data from most recent error
  FOR EACH lxErr IN laErr
    lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
  ENDFOR
  MESSAGEBOX(lcErrMsg)
ENDIF
```

As you can see from this code listing, the query is executed by calling the SQLEXEC command. This form of SQLEXEC takes three parameters. The first parameter is the handle to the open connection that was created in the LOAD event through the call to SQLCONNECT.

The second parameter of SQLEXEC is the query to be executed. In this case, the query is whatever text you have assigned to the m.lcSQLStatement string variable. The third parameter is the name of the cursor that will be returned if there is at least one result set produced by the query. In this case, RUNQUERY always assigns the result table to the cursor name "curResults".

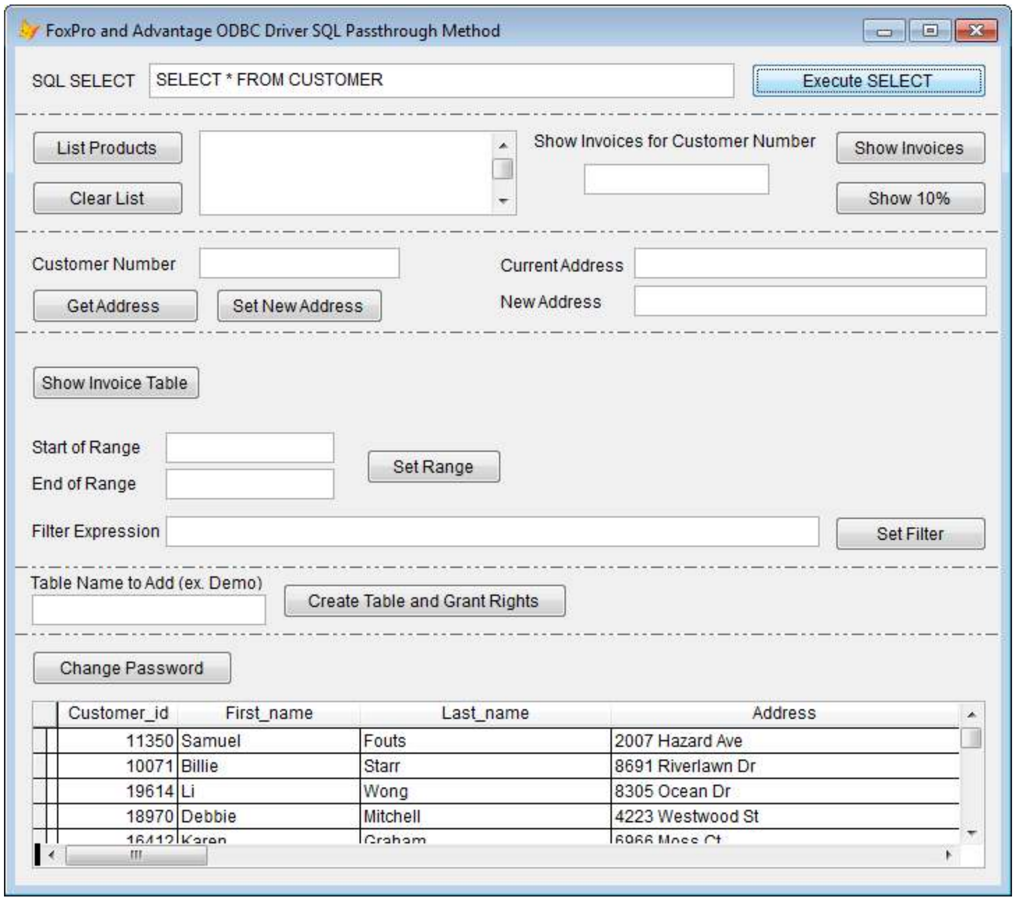
SQLEXEC returns an integer indicating the number of results sets returned by the query. In this project, that value should be either 1 or 0. If SQLEXEC returns -1, an error has occurred, in which case the AEEROR function is called to retrieve the last error, which is display displayed using the MESSAGEBOX function.

If at least one result set is returned, the cursor name is assigned to the RecordSource property of a grid named grResult, which causes the data returned in this result set to be displayed in the this grid, which appears at the bottom of the main form. When no result set is returned, the grid's RecordSource is set to an empty string, which clears the grid.

An example of using RUNQUERY can be found in the Click event associated with the button labeled Execute SELECT. This button executes the SQL statement that you enter into the Text Box named txtSelect. The following is the click event associated with the Button labeled Execute SELECT:

```
IF EMPTY(thisform.TxtSelect.Text)
  MESSAGEBOX("Please enter a SQL statement")
  RETURN
ENDIF
* This provides adHoc query capability
m.lcSQLStatement = thisform.TxtSelect.Text
thisform.runquery
```

Figure 20-1 depicts the main form after the Execute SELECT button was clicked after enter a simple SELECT * FROM CUSTOMER query.



SQL SELECT: SELECT * FROM CUSTOMER

Customer_id	First_name	Last_name	Address
11350	Samuel	Fouts	2007 Hazard Ave
10071	Billie	Starr	8691 Riverlawn Dr
19614	Li	Wong	8305 Ocean Dr
18970	Debbie	Mitchell	4223 Westwood St
16412	Karen	Graham	6966 Moss Ct

Figure 20-1: The main form

Reading and Writing Data

When using SQL pass-through, you read and write data using SELECT and UPDATE queries. For example, reading a single record is demonstrated by the Click event associated with Get Address button (shown in Figure 20-1), shown in the following code segment:

```
IF EMPTY(thisform.txtcustNum.Text)
  MESSAGEBOX("Please enter a customer number")
  RETURN
ENDIF
* Select the product data using an SQL statement
lcErrMsg = "Error"
m.lcSQL = "SELECT * FROM Customer WHERE [Customer ID] = " + ;
  thisform.txtcustNum.Text
m.liX = SQLEXEC(m.nConn, m.lcSQL, "csrCust")
if m.liX > 0
 * display
 if used("csrCust") and reccount("csrCust") > 0
```

```
  * only one record should be returned
  SCAN
    thisform.txtCurrentAddress.Value = csrCust.Address
  ENDSCAN
 ELSE
  * no results returned
  thisform.txtCurrentAddress.Value = "Customer Not Found"
 ENDIF
ELSE
  * display error
  AERROR(laErr) && Data from most recent error
  FOR EACH lxErr IN laErr
    lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
  ENDFOR
  MESSAGEBOX(lcErrMsg)
  *thisform.lstResults.AddItem("Error: " + m.lcErrMsg)
ENDIF
```

The Click event associated with the button labeled Set New Address demonstrates a write operation. This event is shown in the following code segment:

```
IF EMPTY(thisform.txtCustNum.Text)
  MESSAGEBOX("Please enter a customer number")
  RETURN
ENDIF
IF EMPTY(thisform.txtNewAddress.Text)
  MESSAGEBOX("Please enter a new address")
  RETURN
ENDIF
lcErrMsg = "Error"
m.lcSQL = "UPDATE Customer SET Address = '" + ;
  thisform.txtNewAddress.Text + ;
  "' WHERE [Customer ID] = " + thisform.txtcustNum.Text
m.liX = SQLEXEC(m.nConn, m.lcSQL, "csrCust")
if m.liX > 0
 MESSAGEBOX("Address Updated")
ELSE
  * display error
  AERROR(laErr) && Data from most recent error
  FOR EACH lxErr IN laErr
    lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
  ENDFOR
  MESSAGEBOX(lcErrMsg)
  *thisform.lstResults.AddItem("Error: " + m.lcErrMsg)
ENDIF
```

Using a Parameterized Query

Although you can assemble queries at runtime based on data entered into the user interface by concatenating strings, including the user's input, parameterized queries offer a better solution. There are two advantages to parameterized queries. First, when the same basic query needs to executed more than once, but with slightly different values, a parameterized query permits the query to be validated once, but run many times.

A second and more important advantage to parameterized queries is that they prevent a kind of hack referred to as SQL injection. SQL injection, which is described in detail in Chapter 12, *Introduction to Using Advantage SQL*, allows a user to change the operation of a query, often producing unwanted and even damaging results.

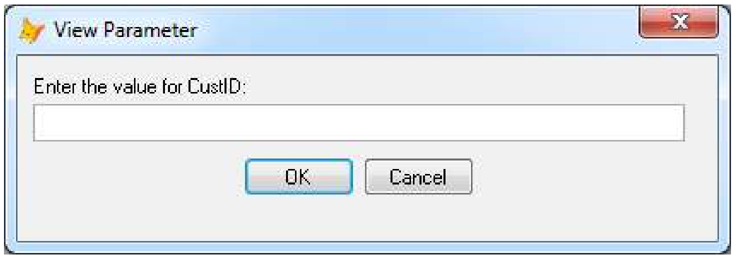
Using SQL pass-through you include one or more parameters in a query by adding a question mark (?) followed by a parameter name. These parameters most often appear in the WHERE clause of a query, but can also appear in the HAVING clause.

If you do not provide a value for the parameter, Visual FoxPro will display a dialog box asking for a value from the user when you attempt to execute the query.

For example, consider the following code snippet:

```
m.lcSQL = "SELECT [Invoice No] as InvcNo ;
   FROM Invoice WHERE [Customer ID] = ?CustID"
m.liX = SQLEXEC(m.nConn, m.lcSQL, "csrInvoice")
```

This code, which is associated with the button labeled Show Invoices (2), produces the following dialog box when the SQLEXEC function is executed.



Enter the value for CustID:

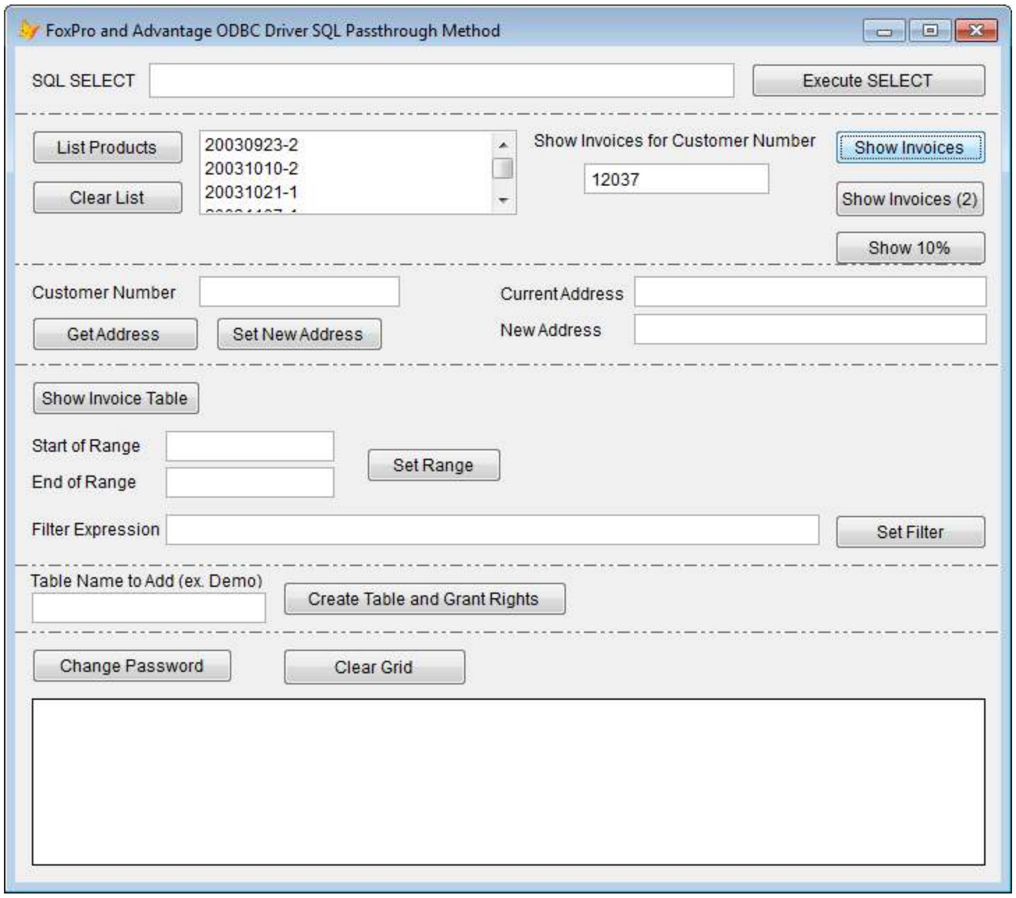
If there are two or more parameters in the query, one dialog box is displayed for each parameter.

While this approach works, it is usually best if you define the parameter programmatically, instead of relying on the user to enter the parameter at the time that the query is executed. This gives your application more control, and even permits you to display customized prompts to the user.

To do this, simply assign a value to the parameter prior to defining the query. This is demonstrated in the code associated with the button labeled Show Invoices. In this Click event, the value of the CustID parameter is retrieved from the text box named txCustID prior to the definition of the query, as shown in this code segment:

```
CustID = thisform.txtcustID.Text
m.lcSQL = "SELECT [Invoice No] as InvcNo ;
 FROM Invoice WHERE [Customer ID] = ?CustID"
m.liX = SQLEXEC(m.nConn, m.lcSQL, "csrInvoice")
```

Since the value of the CustID parameter is already assigned when the query is defined, no dialog box is displayed when the SQLEXEC function is called. Instead, the invoice records for the specified customer are retrieved and inserted into the provided list box, as shown in Figure 20-2.



20030923-2
20031010-2
20031021-1

Show Invoices for Customer Number 12037

Figure 20-2. A parameterized query retrieved its value from a text box

The following is the entire Click event handler associated with the button labeled Show Invoices:

```
IF EMPTY(thisform.txtCustID.Text)
  MESSAGEBOX("Please enter a customer number")
  RETURN
ENDIF
* Clear the list view
thisform.lstResults.Clear()
NOTE Select the product data using an SQL statement
lcErrMsg = "Error"
CustID = thisform.txtcustID.Text
m.lcSQL = "SELECT [Invoice No] as InvcNo ;
 FROM Invoice WHERE [Customer ID] = ?CustID"
m.liX = SQLEXEC(m.nConn, m.lcSQL, "csrInvoice")
if m.liX > 0
 * display
 if used("csrInvoice") and reccount("csrInvoice") > 0
  * loop through the cursor
  SCAN
    thisform.lstResults.AddItem(csrInvoice.InvcNo)
  ENDSCAN
 ELSE
  * no results returned
  thisform.lstResults.AddItem("No Invoices Found.")
 ENDIF
ELSE
  * display error
  AERROR(laErr) && Data from most recent error
  FOR EACH lxErr IN laErr
    lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
  ENDFOR
  MESSAGEBOX(lcErrMsg)
  *thisform.lstResults.AddItem("Error: " + m.lcErrMsg)
ENDIF
```

Calling a Stored Procedure

Stored procedures are invoked using a SQL EXECUTE PROCEDURE query. In every other way, invoking stored procedures is no different than executing any other query. For example, you invoke the stored procedure by calling the SQLEXEC function. If the stored procedure returns a result set, you access it using a named cursor.

In addition, just as regular queries can make use of parameters, so can EXECUTE PROCEDURE calls. Specifically, you can pass one or more of a stored procedure's parameters using query parameters. Simply precede the parameter in the procedure call with a question mark. If you do not explicitly assign a value to the parameter before attempting to execute the stored procedure, Visual FoxPro will display a dialog in order to get the parameter value from the user.

Calling a stored procedure is demonstrated by the code associated with the Click event for the button labeled Show 10%. This particular stored procedure call makes use of a parameterized query, where the parameter value is retrieved from the txCustID text box, as shown in the following code segment:

```
* Clear the list view
thisform.lstResults.Clear()
NOTE Select the product data using an SQL statement
lcErrMsg = "Error"
CustID = thisform.txtcustID.Text
m.lcSQL = "EXECUTE PROCEDURE SQLGet10Percent( ?CustID )"
m.liX = SQLEXEC(m.nConn, m.lcSQL, "csrInvoice")
if m.liX > 0
 * display
 if used("csrInvoice") and reccount("csrInvoice") > 0
  * loop through the cursor
  SCAN
    thisform.lstResults.AddItem(csrInvoice.InvoiceNo)
  ENDSCAN
 ELSE
  * no results returned
  thisform.lstResults.AddItem("No Invoices Found.")
 ENDIF
ELSE
  * display error
  AERROR(laErr) && Data from most recent error
  FOR EACH lxErr IN laErr
    lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
  ENDFOR
  MESSAGEBOX(lcErrMsg)
ENDIF
```

Navigational Actions with Visual FoxPro and SQL Pass-through

While Visual FoxPro offers a number of functions for navigational support, these apply to local tables. When using SQL pass-through, you are generally working with cursors returned from the execution of SQL statements against the remote server. As a result, SQL pass-through offers little in the way of navigational support. In fact, there is one type of navigational support, and that is for scanning a cursor returned from a SQLEXEC statement.

Still, some of the features that we often associate with navigational support, such as filters and ranges, can be achieved in SQL pass-through using the WHERE clause in a SQL statement. As a result, those particular operations are also described in this section.

One final note about navigation is worth mentioning here. In many cases, navigational operations involve one or more of a table's indexes. When using SQL pass-through, indexes are an important factor in the speed of queries against your Advantage data. However, these queries are entirely server-side. In other words, your Advantage indexes are used by ADS to quickly select and join your data. However, once that data is returned to the client application through a cursor, indexes are no longer involved. In other words, the actual cursors return by queries do not have indexes.

Scanning a Cursor

Scanning is the process of sequentially reading every record in a result set, or every record in the range and/or filtered view of the result set. With SQL pass-through, scanning involves retrieving a result set from a query, followed by repeated calls to advance one record at a time until all of the records have been visited.

Although scanning is a common task, it is important to note that it necessarily requires the client application to retrieve all of the records in the result set. This is not a problem when few records are involved, but if a large number of records are being scanned, network resources may be taxed.

Tip: If you are using ADS, and you must scan a large number of records, implement the operation using a stored procedure as described in Chapter 7, "Creating Stored Procedures." Scanning from a stored procedure executed from ADS requires no network resources when the server and the data are located on the same machine.

The following code demonstrates scanning with a cursor. This code, associated with the Click event of the button labeled List Products (shown in Figure 20-1), navigates the entire PRODUCTS table, assigning data from each record to the lstResults list box:

```
NOTE Select the product data using an SQL statement
lcErrMsg = "Error"
m.liX = SQLEXEC(m.nConn, "SELECT [Product Name] as ProdName ;
   FROM Products", "csrProd")
if m.liX > 0
 * display
 if used("csrProd") and reccount("csrProd") > 0
  * loop through the cursor
  SCAN
    thisform.lstResults.AddItem(csrProd.ProdName)
  ENDSCAN
 ELSE
  * no results returned
  thisform.lstResults.AddItem("No Results.")
 ENDIF
ELSE
  * display error
  AERROR(laErr) && Data from most recent error
  FOR EACH lxErr IN laErr
    lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
  ENDFOR
  MESSAGEBOX(lcErrMsg)
  *thisform.lstResults.AddItem("Error: " + m.lcErrMsg)
ENDIF
```

Finding a Record Based on Data

The WHERE predicates of your queries are used to locate specific records, which are then returned to Visual FoxPro in a cursor. While not the same as a seek, they do have the desired effect of retrieving only that record or those records that you are interested in.

Here is another way to look at it. The WHERE clause of a query permits you to perform a task similar to a seek, filter, or range.

For example, the following query, associated with button labeled Get Address, which returns exactly one record (if there is a match) or no records. As a result, this query is similar to a SEEK operation:

```
IF EMPTY(thisform.txtcustNum.Text)
  MESSAGEBOX("Please enter a customer number")
  RETURN
```

```
ENDIF
* Select the product data using an SQL statement
lcErrMsg = "Error"
m.lcSQL = "SELECT * FROM Customer WHERE [Customer ID] = " + ;
  thisform.txtcustNum.Text
m.liX = SQLEXEC(m.nConn, m.lcSQL, "csrCust")
if m.liX > 0
 * display
 if used("csrCust") and reccount("csrCust") > 0
  * only one record should be returned
  SCAN
    thisform.txtCurrentAddress.Value = csrCust.Address
  ENDSCAN
 ELSE
  * no results returned
  thisform.txtCurrentAddress.Value = "Customer Not Found"
 ENDIF
ELSE
  * display error
  AERROR(laErr) && Data from most recent error
  FOR EACH lxErr IN laErr
    lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
  ENDFOR
  MESSAGEBOX(lcErrMsg)
  *thisform.lstResults.AddItem("Error: " + m.lcErrMsg)
ENDIF
```

By comparison, the code shown in the following code segment returns a set of records. In this respect it is somewhat similar to a range or a filter, the resulting records being controlled by the WHERE clause in the query. This code is associated with the Click event of the button labeled Set Range:

```
IF EMPTY(thisform.txtStartRange.Text)
  MESSAGEBOX("Please enter the start of the range")
  RETURN
ENDIF
IF EMPTY(thisform.txtEndRange.Text)
  MESSAGEBOX("Please enter the end of the range")
  RETURN
ENDIF
Startval = thisform.txtStartRange.text
Endval = thisform.txtEndRange.text
m.nResult = SQLEXEC(m.nConn, "SELECT * FROM Invoice ;
  WHERE [Invoice No] BETWEEN ?StartVal AND ?EndVal", "curResults")
IF m.nResult > 0
  thisform.grResult.RecordSource = "curResults"
ELSE
  * display error
  AERROR(laErr) && Data from most recent error
  FOR EACH lxErr IN laErr
    lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
  ENDFOR
  MESSAGEBOX(lcErrMsg)
ENDIF
```

Administrative Operations with Advantage and Visual FoxPro

While Advantage requires little in the way of periodic maintenance to keep it running smoothly, many applications need to provide administrative functionality related to the management of users, groups, and objects.

This section is designed to provide you with insight into exposing administrative functions in your client applications. Two related, yet different, operations are demonstrated here. In the first, a new table is added to the database and all groups are granted access rights to it. This operation requires that you establish an administrative connection (or a user connection that has the appropriate WITH GRANT privileges).

The second operation involves permitting individual users to modify their own passwords. Especially in the security-conscious world of modern database management, this feature is often considered an essential step to protecting data.

Creating a Table and Granting Rights to It

The code associated with the Click event of the button labeled Create Table and Grant Rights permits a user to enter the name of a table that will be created in the data dictionary, after which, all groups, with the exception of the default groups, such as DB:Public, will be granted rights to the table. This Click event is shown in the following code segment:

```
lcErrMsg = "Error"
IF EMPTY(thisform.txttableName.Text)
  MESSAGEBOX("Enter a name for the new table")
  RETURN
ENDIF
* create an admin connection for this task
* this next function call assumes that adssys has no password
m.lnAdminConn = SQLCONNECT("DemoADD", "Adssys")
TRY
  * make sure the table doesn't already exist in the dictionary
  m.lcSQL = "SELECT COUNT(*) as Tblcnt FROM system.tables " + ;
    "WHERE UCASE(Name)" +;
    " = '" + UPPER(ALLTRIM(thisform.txttableName.Text)) + "'"
  m.lnReturn = SQLEXEC(m.lnAdminConn, m.lcSQL, "curResult")
  IF m.lnReturn > 0
    SKIP 1 IN 'curResult'
    IF curResult.Tblcnt <> 0
      MESSAGEBOX("Table already exists")
      RETURN
    ENDIF
  ELSE
    * display error
    AERROR(laErr) && Data from most recent error
    FOR EACH lxErr IN laErr
      lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
    ENDFOR
    MESSAGEBOX(lcErrMsg)
    RETURN
  ENDIF

  * create a table with the specified name
  m.lcSQL = "CREATE TABLE " + ALLTRIM(thisform.txttableName.Text) ;
    + " ( [Full Name] Char(30), [Date of Birth] DATE," ;
    + " [Credit Limit] MONEY, Active LOGICAL)"
  m.lnReturn = SQLEXEC(m.lnAdminConn, m.lcSQL)
  IF m.lnReturn < 0
    * display error
    AERROR(laErr) && Data from most recent error
    FOR EACH lxErr IN laErr
      lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
    ENDFOR
    MESSAGEBOX(lcErrMsg)
    RETURN
  ENDIF

  * Get the names of all users
  m.lcSQL = "SELECT Name FROM " + ;
    "system.usergroups WHERE Name NOT LIKE 'DB:%'"

  m.luReturn = SQLEXEC(m.lnAdminConn, m.lcSQL, "curUsers")
  IF m.luReturn < 0
    * display error
    AERROR(laErr) && Data from most recent error
    FOR EACH lxErr IN laErr
      lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
    ENDFOR
    MESSAGEBOX(lcErrMsg)
    RETURN
  ELSE
    SCAN
      m.lcSQL = "GRANT ALL ON " + ;
      ALLTRIM(thisform.txttableName.Text) + ;
      " TO " + ALLTRIM(curUsers.Name)
      m.lnReturn = SQLEXEC(m.lnAdminConn, m.lcSQL)
      IF m.lnReturn < 0
        * display error
        AERROR(laErr) && Data from most recent error
        FOR EACH lxErr IN laErr
          lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
        ENDFOR
        MESSAGEBOX(lcErrMsg)
        RETURN
      ENDIF
    ENDSCAN
  ENDIF
FINALLY
  * close the admin connection
  SQLDISCONNECT(m.lnAdminConn)
ENDTRY
```

This code begins by verifying that the user has supplied a name for the new table. Next, if performs a query to determine that the table is not already in the data dictionary. At this point, the new table is created using the CREATE TABLE SQL statement.

Once the table has been created, this code queries the system.usergroups system table to get a list of all groups, with the exception of the default groups. ALL rights are then granted to each group returned by this query.

Note: The administrative user name and passwords (blank in this case) are represented by string literals in this code segment. This was done for convenience, but in a real application, either you would ask for this information from the user or you would encrypt this data to prevent its discovery. (String literals can be inspected in compiled applications using a variety of tools.)

Changing a User Password

A user can change the password on the user's own connection, if you permit this change. In most cases, only when every user has a distinct user name would you expose this functionality in a client application. When multiple users share a user name, this operation is usually reserved for an application administrator.

The following method, associated with the button labeled Change Password button (shown in Figure 20-1), demonstrates how you can permit a user to change their password from a client application:

```
* setup variables
m.lcOldPass = ""
m.lcNewpass1 = ""
m.lcNewpass2 = ""
m.lcUserName = ""
lcErrMsg = "Error"

* get current password
m.lcOldPass = INPUTBOX("Enter your current password")
IF ISBLANK(m.lcOldPass)
  MESSAGEBOX("Password cannot be blank")
  RETURN
ENDIF

* verify that this password is good
* begin by getting the user's username
m.lcSQL = "SELECT USER() as UserName FROM system.iota"
m.liX = SQLEXEC(m.nConn, m.lcSQL, "crsUser")
* only one record will be returned
SCAN
  m.lcUserName = crsUser.UserName
ENDSCAN
* now, create a new connection using the username and password
validPassword = .T.
TRY
```

```
  SQLSETPROP(nStatementHandle, 'DispLogin', 3)
  m.lnTestConn = SQLCONNECT("DemoADD", m.lcUserName, m.lcOldPass, .T.)
CATCH
  validPassword = .F.
  MESSAGEBOX("Cannot confirm existing password")
ENDTRY
IF NOT validPassword
   RETURN
ENDIF

* verify the password
m.lcNewpass2 = INPUTBOX("Confirm your new password")
IF m.lcNewpass1 == m.lcNewpass2
  * change the password
  m.lcSQL = "EXECUTE PROCEDURE sp_ModifyUserProperty( USER, ;
   USER_PASSWORD', '" + ALLTRIM(m.lcNewpass1) + "' )"
      + ALLTRIM(m.lcNewpass1) + "' )"
  m.lnReturn = SQLEXEC(nConn, m.lcSQL, "curResult")
  IF m.lnReturn < 0
    * display error
    AERROR(laErr) && Data from most recent error
    FOR EACH lxErr IN laErr
      lcErrMsg = lcErrMsg + TRANSFORM( lxErr, "")
    ENDFOR
    MESSAGEBOX(lcErrMsg)
  ENDIF
  MESSAGEBOX("You have successfully changed your password")
ELSE
  MESSAGEBOX("Passwords do not match")
  RETURN
ENDIF
```

This code segment is a bit simpler than the preceding one. After verifying that the user knows the current password with which they are connected (by attempting to create a new connection using the user's user name and the entered old password), they are prompted for their new password twice, for confirmation purposes. Next, the sp_ModifyUserProperty system stored procedure is called to change the password. Finally, a dialog box reports to the user whether they were successful or not.

Note: If you run this code, and change the password of the adsuser account, you should use the Advantage Data Architect to change the password back to password. Otherwise, you will not be able to run this project again, since the literal 'password' is passed to the SQLCONNECT function in the form's LOAD event.