CHAPTER 16

Working with Recordsets—Part III

In This Chapter

Updating fields in a recordset

Adding and deleting records in a recordset

Batching changes

Using transactions

Working with disconnected recordsets

Working with other recordset functions

In this chapter, I'll wrap up my coverage of the ADO `Recordset` object by discussing how to update the information in a particular record, and then I'll explain other more advanced ideas such as batch updates and transactions. I'll conclude the chapter by covering how to clone recordsets, resync, and requery recordsets, and how to return multiple recordsets. I'll also show you some alternate ways to get data, and finally, how to set the cache size for recordsets.

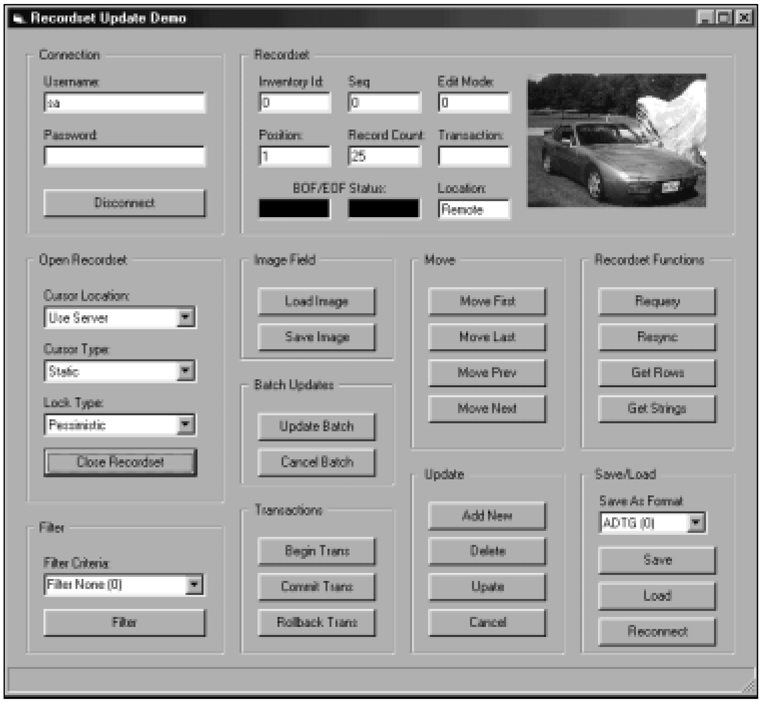
Up until this point, I've focused on how to move the current record pointer around in a `Recordset` object and how to access individual fields in it. Now it's time to discuss how to change the information in a recordset. You can insert, delete, and update the individual records in a recordset by using the `AddNew`, `Delete` and `Update` methods.

Updating Recordsets

Updating records in a `Recordset` can involve no more than selecting the appropriate `LockType` when you open the recordset. After the user has updated the field using bound controls or with code in your program that explicitly changes a field's `Value` property, using any of the move methods will automatically save the changes. However, having a deeper understanding of how the update process works will help you fine-tune your application to run faster and better.

Recordset Update DemoThe Recordset Update Demo program (see Figure 16-1) is designed to let you try different approaches to updating a recordset while watching the various indicators.

While this program is similar to the Recordset Movement Demo program (and uses much of the same code), it is more complicated to use. In the Recordset Movement Demo program, most of the activities require only one button click in order to perform a function. The Recordset Update Demo program usually requires a number of separate steps in order to accomplish a task successfully. This process merely reflects the number of individual steps it takes to update a database record.



Username: sa
Password:
Inventory Id: 0
Seq: 0
Edit Mode: 0
Position: 1
Record Count: 25
Transaction:
BOF/EOF Status:
Location: Remote
Cursor Location: Use Server
Cursor Type: Static
Lock Type: Pessimistic
Filter Criteria: Filter None (0)
Save As Format: ADTG (0)

Figure 16-1: Running the Recordset Update Demo program

On the CD-ROM

You can find the Recordset Update Demo program on the CD-ROM in the `\VB6DB\Chapter16\RecordsetUpdateDemo` directory.

Like the Recordset Movement Demo program, the Recordset Update Demo program relies on bound controls to display information from the database. I used a different table in this program to show you how to use an image to display a picture from an Image field in the table. Unfortunately, you can only display an image from the database with the `Image` control. You must manually load the image file from disk to update the value in the database. See Chapter 15 for more details about how to use the `AppendChunk` method to load a file to the **Image** field.

In a typical update scenario, you press the Add New button to create a new record in the database and then fill in values for the two bound fields (InventoryId and Seq). Next, you press the Load Image button to fill in a value for the image field. Finally, you press the Update button to save the image to the database. Of course, this simple example assumes that you have opened the recordset with a cursor that can be updated.

Tip

I did it first: I often find myself writing small programs that allow me to explore the various features of an object. The program usually starts out simple and often ends up fairly complex. The Recordset Update Demo and the Recordset Movement Demo programs provide examples of applications that I wrote to test various features in ADO. Rather than write your own small programs to try combinations of things in ADO, I suggest you use these programs instead. They'll save you a lot of time and energy.

Updating an existing record

The update process begins as soon as you move the current record pointer to a new row. If you selected pessimistic locking when you opened the recordset, the database server will immediately place a lock on the row, preventing any other user from accessing the row until you have finished working with it.

Note

The read-only and the batch: This section addresses only what happens when you use pessimistic or optimistic locking with a server-side cursor. Obviously, you can't update a recordset that was opened in read-only mode, and I'll cover batch optimistic locks and client-side cursors later in this chapter under "Performing batch updates" and "Working with Disconnected Recordsets," respectively.

When the row is retrieved from the database server, its values will be made available to you in the `Fields` collection of the `Recordset` object. As you saw in Chapter 15, each `Field` object in the collection keeps the original value of the field as it was retrieved separate from the current value as seen by your program. The current value is stored in the `Value` property, while the original value is stored in the `OriginalValue` property. A third value, which is available on demand, allows you to retrieve the value of the field as it currently exists in the database server at that instant. The `UnderlyingValue` property is useful only with optimistic locking, since it is impossible for the value of the field to change while the row is locked using pessimistic locking.

Note

Binding effects: Bound controls work exactly the same with `Recordset` objects as they do with the `ADO Data Control`. When the user changes a value in a bound control, the `Value` property of the `Field` object will be changed automatically. This action triggers all of the associated processes for the `Recordset`.

Initially, the EditMode property will be set to adEditNone (0), which means that no changes have occurred to the record. The moment one of the values changes, the EditMode property will be set to adEditInProgress (1), which means that one or more values of the current record have been changed.

Once you've finished changing the values in the current row, you can decide whether to save your changes or to discard them. In database terms, you either perform a *commit* to save your changes or a *roll back* to undo all of your changes. If you choose to roll back your changes, you must call the Cancel method (see Listing 16-1). This method takes care of undoing your changes in the database. While it is unlikely, it is possible that you may get a run-time error when using the Cancel method.

Listing 16-1: **The Command12_Click event in Recordset Update Demo**

```
Private Sub Command12_Click()

On Error Resume Next

Images.Cancel
If Err.Number <> 0 Then
   WriteError

End If

Text10.Text = FormatNumber(Images.EditMode, 0)

End Sub
```

Updating the data is a lot more complex than canceling the update, even though the code is similar (see Listing 16-2). What happens under the covers depends on the locking mode you chose. If you chose pessimistic locking, the values will be returned to the database server and applied to the database. The EditMode property will be reset to adEditNone. The lock on the record at the database server will remain in place until you move to a new row.

If you chose optimistic locking, a series of activities will take place. First, a lock is placed on the row at the database server. Then, for each field in the recordset's Fields collection, the OriginalValue property will be compared to the UnderlyingValue property. If there are any differences, the Update method will be terminated, an error will be returned in the Errors collection and the lock will be released. If the original values are the same as the underlying values, the changed values in the Value property will be saved to the database server. Then the lock on the current row will be released, and EditMode will be reset to adEditNone.

Listing 16-2: **The Command10_Click event in Recordset Update Demo**

```
Private Sub Command10_Click()

On Error Resume Next

Images.Update
If Err.Number <> 0 Then
   WriteError

End If

Text10.Text = FormatNumber(Images.EditMode, 0)

End Sub
```

No matter which locking strategy you chose, it is still possible for the update to fail. You may have changed the primary key so that it duplicates another value in the table. It is possible that you may have used a set of values that violates a foreign key constraint, or you may have triggered another constraint. In these situations, the update will not be made to the database, and the `Update` method will fail. Your current data will remain in the `Fields` collection, and you may attempt to correct the problem and try the `Update` method again.

Adding a new record

Adding a new record is similar to updating an existing record, except that no locks are required. The `AddNew` method creates a new database record by creating an empty row, which then becomes the current record (see Listing 16-3). All of the `Field` objects will automatically be set to **Null** or the default value as specified in the table definition. The `EditMode` property will be set to `adEditAdd`, indicating you're editing a newly accessed record. Your program is then free to change the values as desired.

Listing 16-3: **The Command7_Click event in Recordset Update Demo**

```
Private Sub Command7_Click()

On Error Resume Next
```

Continued

Listing 16-3: *(continued)*

```
Images.AddNew
If Err.Number <> 0 Then
   WriteError

End If

Text10.Text = FormatNumber(Images.EditMode, 0)

End Sub
```

When your program is ready to save the new record to the database, you need to call the `Update` method. The new record will be sent to the database server for processing. If you have an identity field, its value will be computed before it is saved to the database. If there are any errors with the values in the record, such as a duplicate primary key, invalid foreign key reference, not **Null** constraints, or any other constraint violations, an error will be returned, and the record won't be added. However, the values will remain in the `Fields` collection, and you can correct them to try again. Alternately, you can call the `Cancel` method to undo the `AddNew` method.

The newly added record may or may not be visible in the `Recordset` depending on your provider and the cursor type. You may have to use the `ReQuery` method to make it visible.

Note

Yet another way to add: If you specify a list of field names and their values as part of the call to `AddNew`, the new record will be created, the listed values will be applied to their associated fields, and the record will be saved to the database. There's no need to call the `Update` method. Note that this method may still fail for the reasons I discussed earlier, at the end of "Updating an existing record."

Deleting an existing record

Deleting the current record is as easy as calling the `Delete` method (see Listing 16-4). Of course, you must lock the record before deleting it, so it is possible for the `Delete` method to fail if you are using optimistic locking. It is also possible for the `Delete` method to fail if deleting the record would trigger a constraint.

Once the record is deleted, it remains the current record, although if you try to access any of the `Field` objects, a run-time error will occur. As soon as you move the current record pointer to another record, the deleted record will no longer be visible. Once a record has been deleted, you can't undelete it using the `Cancel` method.

Listing 16-4: The Command8_Click event of Recordset Update Demo

```
Private Sub Command8_Click()

On Error Resume Next

Images.Delete
If Err.Number <> 0 Then
   WriteError

End If

Text10.Text = FormatNumber(Images.EditMode, 0)

End Sub
```

The `Delete` method has an interesting capability that allows you to delete multiple rows with a single call. You can delete all of the rows in the `Recordset`, or you can delete only those records selected by the `Filter` property. You can also delete the rows in the current chapter of a hierarchical recordset.

Performing batch updates

The idea behind optimistic locking is that with enough data, the odds that two or more people would try to access the same record are so low as to be nonexistent. Suppose you have an application that falls into this group that performs operations on groups of records rather than just individual records. Perhaps, you have a table that you use to hold the line items entry from an order. So if someone is likely to access one of these rows, that person is likely to access all of them. The same argument could hold true for the course records in a student information system, the transaction records against a single financial account, or the payroll records of a single employee. Each of these groups could be considered a batch, which is updated as a single unit.

ADO supports the concept of batch updating with a *batch optimistic* locking strategy. It uses the same locking strategy that is used for an optimistic lock, but it allows you to create a group of changes that are sent to the database server for processing at a single time. By transmitting the group of changes as a single group, you can get better response time because you don't have to wait on each individual row to be updated. Also, this means less work for the server as it can update the group of rows more intelligently.

In terms of using batch optimistic locks in your application, you need to open the Recordset object with LockType equal to adLockBatchOptimistic. Next you should set CursorLocation to adUseClient and CursorType to adStatic. While other combinations of these properties may work, these will give you the best results. All of the data will be buffered locally in the client computer, which will give you the best performance because you only communicate with the database server at the beginning and the end of the process.

After you open your Recordset, you can use it as you normally would. You can add new records, delete existing ones, or change any values you choose. However, as you make these changes, they are held and not transmitted to the database server for processing. Instead, they are held locally until you explicitly commit them or discard them.

You can discard your changes with the CancelBatch method (see Listing 16-5), or you can save the changes with the UpdateBatch method (see Listing 16-6). However, since the changes are applied with an optimistic lock, you need to check for errors to see if all of your changes were successfully applied. Even if you use the CancelBatch method, it is possible for you to get a warning or an error if someone else deleted some or all of the records you would have updated before you canceled the update.

Listing 16-5: The Command13_Click event in Recordset Update Demo

```
Private Sub Command13_Click()

On Error Resume Next

Images.CancelBatch
If Err.Number <> 0 Then
   WriteError

End If

Text10.Text = FormatNumber(Images.EditMode, 0)

End Sub
```

You can check the status of your data after you use the CancelBatch or the UpdateBatch by setting the Filter property to adFilterConflictingRecords. The filter limits the recordset so that only the records that couldn't be updated during the call to UpdateBatch are visible. You should then scan through the entire recordset and check the Status property to learn why the records weren't properly updated.

Listing 16-6: **The Command11_Click event in Recordset Update Demo**

```
Private Sub Command11_Click()

On Error Resume Next

Images.UpdateBatch
Images.Filter = adFilterConflictingRecords
Images.MoveFirst
Do While Not Images.EOF
   If Images.Status <> 0 Then
      MsgBox "Error " & FormatNumber(Images.Status,0) & _
         " in: " & _
         FormatNumber(Images.Fields("InventoryId").Value, 0)

   End If

   Images.MoveNext

Loop

Text10.Text = FormatNumber(Images.EditMode, 0)

End Sub
```

Making Transactions

An important feature in ADO is the ability to use *transactions* while processing your data. They allow you to identify a series of individual database operations all of which must be successful or all of their effects must be completely removed from the database.

Why do I need transactions?

Think back to Chapter 14 where I was discussing why locking is necessary in a database system. I discussed an example where you needed to serialize the debits and the credits to a bank account to prevent an invalid value from being created. Yet you could create an invalid amount in another way, even with proper locking.

Consider what happens when you transfer money from one account to another. You withdraw money from one account and credit it to another. With locking, you can make sure that each operation ensures that the appropriate values are created in both tables. However, what happens if one operation happens and the other doesn't. If you do the withdrawal first, the money will disappear if it isn't properly credited, while if the credit is made first, the money appears to come out of nowhere.

This is a situation where you have two database operations that must be successful or both operations must fail together. If only one fails, you have a serious problem. The problems that cause the failure of one operation and not the other can range from a simple application error, through a networking problem, and to a major database server problem. While you might think that this example is a little contrived, there are many situations where you have a series of database operations where you want all of them to succeed or none of them.

The solution to this program is to group the set of operations into a package known as a transaction. A transaction is an *atomic* unit because it can't be subdivided. The entire transaction must be applied to the database, or the entire transaction is aborted.

Yet a transaction need not be limited to a single recordset. Most applications update multiple recordsets as a routine part of their processing. Consider the order portion of the sample database. The Orders table contains information that is specific to the order such as the CustomerId and the date the order was placed. Each item in the order has a separate record in the OrderDetails table. To place an order, you have to add one record to the Orders table and one or more records to the OrderDetails table. This is another situation where you want both tables properly updated or neither.

ADO and transactions

Transactions are implemented using the `Connection` object and not the `Recordset` object. You can involve any number of `Recordset` objects and operations as long as they are all using the same `Connection` object. Of course, the single connection limitation implies that a transaction can't span more than one database server, though you may be able to access multiple databases on a single database server.

A transaction involves three basic methods. The `BeginTrans` method (see Listing 16-7) marks the beginning of a transaction. The `CommitTrans` method (see Listing 16-8) marks the successful conclusion of a transaction, while the `RollbackTrans` method (see Listing 16-9) discards all of the changes made and leaves the database untouched.

Note

Transactions aren't batch locks: Don't confuse batch optimistic locks with transactions. They serve two different purposes. Batch optimistic locks are a performance enhancement that can be used successfully on large databases where you don't expect multiple users to try to access the same group of rows. Transactions are used to ensure that the database remains consistent. They can be used in any situation where you have a group of database operations that must be performed together or not at all.

Listing 16-7: **The Command21_Click event in Recordset Update Demo**

```
Private Sub Command21_Click()

On Error Resume Next

db.BeginTrans
If Err.Number <> 0 Then
   WriteError

Else
   Text11.Text = "Active"

End If

End Sub
```

Listing 16-8: **The Command20_Click event in Recordset Update Demo**

```
Private Sub Command20_Click()

On Error Resume Next

db.CommitTrans
If Err.Number <> 0 Then
   WriteError

Else
   Text11.Text = "Committed"

End If

End Sub
```

Listing 16-9: **The Command19_Click event in R ecordset Update Demo**

```
Private Sub Command19_Click()

On Error Resume Next
db.RollbackTrans
If Err.Number <> 0 Then
```

Continued

Listing 16-9 *(continued)*

```
   WriteError
Else
   Text11.Text = "Rollback"

End If

End Sub
```

If you set the `Connection` object's `Attribute` property to either `adXactAbortRetaining` or `adXactCommitRetaining` or both, a new transaction will be started automatically when you call either the `RollbackTrans` or the `CommitTrans` method. While this can be useful in some types of programs, I like the idea of explicitly marking the beginning and end of a transaction. Not only does this make the code that forms the transaction clearer to the next programmer who may have to modify your program, but it also allows you the freedom to perform database activities outside the scope of a transaction.

Working with Disconnected Recordsets

Client-side cursors let you do more of your database work on your local machine rather than continually communicating with the database server for each and every request. However, it is possible to do all of your work locally if you're willing to do a little extra work and then upload your work to the database server when you're finished.

What is a disconnected recordset? The basic idea behind a *disconnected recordset* is that you make a local copy of the data in the recordset and then break the connection to the database server. You can then perform your updates against the local copy, and when you're finished, you can reconnect to the database server and upload the changes.

The key to making this work is the ability to use a client-side cursor and batch optimistic locking. Using these tools, you would go ahead and process your data normally and all of the changes would be buffered locally until you execute the `UpdateBatch` method to transmit them to the database server. So while you might have a connection to the database server, it's not absolutely necessary since there are no communications traveling between the database client and the database server.

Tip

Disconnected from the net: Disconnected recordsets are ideal for situations where you need to make changes to your database from a laptop that is not permanently attached to your network. You can collect these changes into a single batch while the computer is away from the network and upload them when the computer is reattached to the network.

Making a recordset local

A Recordset object can be saved as a local disk file by using the Save method. You can save the recordset using either the ADTG (Advanced Data Table Datagram) or XML (Extensible Markup Language). ADTG is a Microsoft proprietary format that is somewhat more efficient than the XML format and can handle all types of Recordsets. XML is an open standard supported by multiple vendors, but there are some situations (primarily dealing with hierarchical recordsets) where you may lose some functionality. When saving a Recordset for local processing, either format is fine, though XML would be preferred if you want to share the file with someone else.

Saving the recordset

In the example in Listing 16-10, I constructed a filename using the directory path to the application and the first three characters of the file type described in the combo box. Then I used the value from the ItemData list for the current combo box to determine the format used to save the Recordset.

Listing 16-10: **The Command18_Click event of Recordset Update Demo**

```
Private Sub Command18_Click()

On Error Resume Next

Images.Save App.Path & "\localrs." &
   Left(Combo5.Text, 3), _
   Combo5.ItemData(Combo5.ListIndex)

If Err.Number <> 0 Then
   WriteError

End If

End Sub
```

Note

Expensive words: The process of saving a copy of a Recordset locally so it can be reopened while not connected to the database server is known as *persisting the recordset.*

Cross-Reference

I'll talk about XML in more detail starting with Chapter 21.

Once the Recordset has been saved, you can disconnect it from the database server by setting the ActiveConnection property to Nothing. Then you can close the Recordset object and the Connection object. As long as the recordset is connected to the database, you will continue to operate as before.

Opening the saved recordset

Once you have saved the Recordset as just described, you can reopen it using the file you just saved without specifying a connection to the database server (see Listing 16-11). You'll need a valid instance of a Recordset object in order to open the file. When you code the Open method, you need to specify the name of the file as the Source parameter and adCmdFile for the Options parameter. The file will then be copied by the client cursor library into memory and can be accessed as a normal Recordset object, and you can also use bound controls.

Listing 16-11: **The Command22_Click event of the Recordset Update Demo**

```
Private Sub Command22_Click()

On Error Resume Next

If Images Is Nothing Then
   Set Images = New Recordset

Else
   Images.Close

End If

Err.Clear
Images.Open App.Path & "\localrs." & Left(Combo5.Text, 3), , _
   adOpenStatic, adLockBatchOptimistic, adCmdFile

If Err.Number <> 0 Then
   WriteError

Else
   Text8.Text = "Local"

   Text4.DataField = "InventoryId"
   Set Text4.DataSource = Images
   Text7.DataField = "Seq"
   Set Text7.DataSource = Images
   Image1.DataField = "Image"
   Set Image1.DataSource = Images

End If

End Sub
```

Reconnecting to the database server

Once you've finished updating your recordset locally, you need to reconnect to the database server. This can be a few minutes or a few days after you originally saved the data. If you haven't opened the recordset with batch optimistic locks, you need to close the recordset and reopen it.

With the open copy of the local recordset, you then have to set the `Active Connection` property to a valid connection object (see Listing 16-12). This will restore your connection to the database server. Then you can use the `UpdateBatch` method to send the changed records to the database server. It is very important that you check the results of the `UpdateBatch` method; since it is possible someone else may have updated the records while you were editing the recordset locally.

Listing 16-12: **The Command25_Click event in Recordset Update Demo**

```
Private Sub Command25_Click()

On Error Resume Next

Set Images.ActiveConnection = db
If Err.Number <> 0 Then
   WriteError

End If

Images.UpdateBatch
Images.Filter = adFilterConflictingRecords
Images.MoveFirst
Do While Not Images.EOF
   If Images.Status <> 0 Then
      MsgBox "Error " & FormatNumber(Images.Status,0) & _
         " in: " & _
         FormatNumber(Images.Fields("InventoryId").Value, 0)

   End If

   Images.MoveNext

Loop

End Sub
```

Working with Other Recordset Functions

There are few other functions available for recordsets, which I want to briefly touch on in this chapter. For the most part, these functions aren't used very often, but they can add value to your application in the right situation.

Cloning a recordset

Sometimes it is useful to create an identical copy of a `Recordset` object. This may arise if you need multiple current record pointers in the same recordset or if you want to use two different filters at the same time.

In order to clone a recordset, your provider must support bookmarks. The side effect of this is that a bookmark from one recordset will work in all of its clones as long as no filters have been applied or you haven't executed the `Resync` or `Requery` methods. This has yet another side effect; changes made to one recordset will be seen immediately by the other recordsets.

Resyncing and requerying a recordset

When you access a `Recordset` with a static or forward-only cursor, you can't see the changes that someone else may have made in the database. The `Resync` method allows you to get a fresh copy of the values from the database without executing the query again. This means that the `Resync` method is more efficient.

Of course, `Resync` has its limitations. It won't detect when a new record has been added. If any of the records in the `Recordset` are deleted, an error will be generated in the `Error` collection, and you will need to use the `Filter` property to examine these records to determine how you want to handle them.

The `Requery` method on the other hand forces the database server to re-execute the query that was originally used to retrieve the records from the database. This means that records that have been added, deleted, and updated will be properly reflected in your `Recordset`. Calling `Requery` is the equivalent of calling the `Close` and `Open` methods. Obviously, using `Requery` is more expensive (in terms of resources and time) than using `Resync`, but there are times when it is necessary.

Returning multiple recordsets

A stored procedure has the ability to return multiple recordsets with a single call; however, only one recordset is accessible at a time through the `Recordset` object. You can also specify multiple **Select** statements in a `Command` object to return multiple recordsets.

When you execute the `Command` object, the first recordset is generated, and the server waits to generate the next recordset until you call the `NextRecordset` method. If you close the `Recordset` object before retrieving all of the recordsets, the remaining statements are not executed.

Alternate ways to get data

While you can retrieve information from a recordset using the `Fields` collection, there are a couple of other methods of which you should be aware. They are the `GetRows` method and the `GetString` method. Both return one or more rows of information from the `Recordset` object in a single call.

The `GetRows` method returns a `Variant` containing a two-dimensional array where each row of the array corresponds to a row of information from the recordset and each column of the array corresponds to a column in the recordset. The dimensions of the array are automatically resized to handle the amount of data that you request.

You can also specify the field you want to extract by including the name of the field as a parameter to the method. You can specify an array containing list of fields that you want to retrieve.

The `GetString` method performs the same basic function as the `GetRows` method, but it returns the data as a single `String` rather then the cells of an array. You can specify the delimiter that will be used between columns and the delimiter between rows. For example, you can create a CSV (comma separated value) file by specifying a comma (`,`) as the column delimiter and a carriage return/line feed pair (`vbCrLf`) for the row delimiter.

Both methods start with the current record and return the specified number of records. The `GetRows` method allows you to specify the starting position if your `Recordset` supports bookmarks.

Tip

Comma Separated Value files, the easy way: The `GetString` method makes it really easy to create the data for a CSV file.

Note that with either method, you may get an error number 3021 when reading the last block of data. This error number means that you reached the end of the file; however, this is expected, if you read multiple records at a time. In this case, it means that you requested more records than were available, hence the end of file error.

Setting the cache size

One of the problems when building a client/server database is that all of the communications between the client program and the database server are routed through a network. By default, only one record is transmitted between your application and the database server each time you move the current record pointer. However, this value is tunable by using the `CacheSize` property.

The `CacheSize` determines the number of records that are buffered locally by the OLE DB provider. A value of one means that only one record is quickly available to your program. Increasing this size to more than one means that the provider will keep multiple records locally. This means that the provider will not request additional data from the server until you reference a record outside of the cache. As long as your program requests from within the cache, the provider doesn't need to communicate with the database server. When your program does reference a record outside of the cache, the provider will flush the local cache and fill it again with new records from the database server.

Depending on your program, changing the `CacheSize` property can make a big difference, either good or bad. If you are reading through the `Recordset` sequentially, then you'll get the biggest performance boost, especially if you open the recordset in read-only mode with a forward-only cursor. In many cases, the cost of retrieving records is proportional to the number of calls the provider has to make to the database server. Thus, if you retrieve more records from the database server with each call, the fewer calls you will need to make to satisfy the query.

However, if your program is reading records randomly throughout the recordset, having a large cache is a detriment. The database server must retrieve more records from the database and transmit them over the network, which is work that is essentially wasted.

There is another issue using a large cache, which can cause other problems if you are not careful. Since the provider and server are not in contact until you need to retrieve additional records, it is possible that someone else will have changed or deleted a record after the records were retrieved. Your program will need to be able to handle this situation. For read-only access with a forward-only cursor, this shouldn't be a big problem. But if your program is attempting to update records using a `CacheSize` greater than one, you may have problems. I suggest that you either use batch optimistic locking and handle the errors after the fact or drop the `CacheSize` back to one.

Thoughts on Updating Recordsets

Updating Recordsets is one of the most important functions that a database program PERFORMS. ADO provides a wide range of tools to make this process as painless as possible. Once you get comfortable using the ADO object library to access your data, using other tools will feel downright awkward.

Most of the time when I need to build a simple program to update a database, I just open a Recordset object and then bind a bunch of controls to it to display the information. While there is a little more code involved with this process than using something like the ADO Data Control, I don't mind the extra work because I prefer to open the Recordset after I open the form that uses it.

But while it is easy to update a database using a Recordset, you'll probably be better off using stored procedures and a Command object. Depending on the database server and the OLE DB provider, many Recordset operations have to be translated to explicit SQL Statements that are executed directly against the database. While this extra work isn't much, if you have a high-volume application, anything you can do to reduce work will make your application run better. After all, while machine resources are relatively cheap, someone has to pay for them, and if you don't really need the resources, you shouldn't use them.

Summary

In this chapter you learned the following:

✦ You can change the values in a row by using the Fields collection and then save the new values using the Update method.

✦ You can use the AddNew method to insert a new row and the Delete method to remove a row from a recordset.

✦ You can group changes into batches for better performance.

✦ You can define transactions in your application, in which all of the changes must succeed or all of the changes will be discarded.

✦ You can use disconnected recordsets to access information from the database without having an active connection to the database.

✦ You can perform a number of other useful functions with a Recordset object such as returning multiple recordsets and multiple rows.

✦ ✦ ✦