CHAPTER 14

Working with Recordsets — Part I

✦ ✦ ✦ ✦

In This Chapter

The Recordset object

Exploring various types of locks

Choosing a cursor type

Opening a recordset

✦ ✦ ✦ ✦

Of all of the objects in ADO, the `Recordset` object is the one you'll use the most. It contains the actual data from your database. You can use this object to retrieve, insert, update, and delete information from the database. It can be used as a data source for other controls, just like the ADO Data Control. In fact the ADO Data Control exposes a reference to the `Recordset` object, you can access the information contained in the underlying recordset directly.

This chapter and the next two are dedicated to the topic of recordsets. In this chapter, I'm going to discuss the various properties, methods, and events in a `Recordset` object. Then I'll discuss how to open and use a `Recordset` object. In the next chapter, I'll continue discussing the `Recordset` object and cover how to move around inside recordsets and how to access the individual fields. Finally, in Chapter 16 I'll cover the issues related to updating the data contained in a recordset.

The Recordset Object

The `Recordset` object contains the set of rows retrieved from a database query. Various properties and methods instruct the OLE DB provider on how the rows should be returned from the database and how the provider should handle updates and locking to ensure proper access to the data. Other properties and methods in the `Recordset` object allow you to access the set of rows retrieved.

Note

Birds of a Feather aren't at the first record of the Recordset: There is a common misconception that when BOF is TRUE, you are at the first record of the Recordset. This isn't true. When BOF is TRUE, the current record pointer is before the first record in the Recordset, and there isn't a current record. The same is true for EOF, except in that case, the current record pointer is beyond the last record in the Recordset.

Recordset object properties

Table 14-1 lists the properties associated with the Recordset object. Tables 14-2 through 14-8 contain additional information about individual properties in the Recordset object.

Table 14-1
Properties of the Recordset Object

Property	***Description***
AbsolutePage	A Long value containing the absolute page number of the current record in the Recordset ranging from 1 to PageCount. This value may also be adPosBOF (-2) if the current record is at the beginning of the file or adPosEOF (-3) if the current record is at the end of the file. A value of adPosUnknown (-1) may be used if the Recordset is empty or the provider doesn't support this property.
AbsolutePosition	A Long value containing the absolute position of the current record in the Recordset ranging from 1 to RecordCount. This value may also be adPosBOF (-2) if the current record is at the beginning of the file or adPosEOF (-3) if the current record is at the end of the file. A value of adPosUnknown (-1) may be used if the Recordset is empty or the provider doesn't support this property.
ActiveCommand	An object reference to the Command object that created the Recordset. If the Recordset wasn't created using a Command object, this property will have a **Null** object reference.
ActiveConnection	A Variant array containing either a String value with a valid connection string or an object reference to a Connection object.
BOF	A Boolean value that is TRUE when the current record position is before the first record in the Recordset.

Property	***Description***
Bookmark	A Variant that sets or returns a value that uniquely identifies the current record in the current Recordset object. You can save this property and move to a different record. Then you can restore the value to return back to the original record.
CacheSize	A Long value containing the number of records kept in the provider's cache. The default value for this property is 1.
CursorLocation	An enumerated value that indicates where the cursor is maintained (see Table 14-2).
CursorType	An enumerated value that specifies the type of cursor that will be used on the Recordset. (see Table 14-3).
DataMember	A String containing the name of the data member that will be retrieved using the DataSource property. Works with the DataSource property to bind a Recordset to a control.
DataSource	An object reference that indicates the data source when using bound controls.
EditMode	An enumerated value containing the edit mode of the current record (see Table 14-4).
EOF	A Boolean value that is TRUE when the current record position is after the last record in the Recordset.
Fields	An object reference to a Fields collection containing the fields associated with the current record.
Filter	A Variant value containing one of the following: a String value containing an expression similar to that in a **Where** clause that will select only those records meeting the specified criteria; an Array of bookmarks; or a Long value selected from Table 14-5.
Index	A String containing the name of an index that is used in conjunction with the Seek method.
LockType	An enumerated value containing the locking mode used by the Recordset when retrieving records (see Table 14-6).
MarshalOptions	An enumerated value that indicates how records are to be marshaled back to the server. Applies only to Recordsets with client-side cursors (discussed later in this chapter). The default value is adMarshalAll (0), which returns all rows back to the server for processing, while adMarshalModifiedOnly (1) returns only the modified rows.

Continued

Table 14-1 *(continued)*

Property	***Description***
`MaxRecords`	A `Long` value that specifies the maximum number of rows to be returned. A value of zero implies there is no maximum limit.
`PageCount`	A `Long` value containing the number of pages of data in the `Recordset`. The number of records in a page is specified by the `PageSize` property, while the current page number is specified in the `AbsolutePage` property.
`PageSize`	A `Long` value containing the number of records in a single page.
`Properties`	An object reference to a `Properties` collection containing provider specific information.
`RecordCount`	A `Long` value containing the number of records retrieved. If the provider is not able to determine the number of records retrieved or a forward-only cursor is selected, this property will have a value of –1.
`Sort`	Specifies a list one or more of field names on which the `Recordset` will be sorted. Multiple fields need to be separated by commas, and each field name may be followed by ASC or DESC to sort the fields in either ascending or descending order. The syntax is basically the same as the **Order By** clause in the **Select** statement. This property is only valid when you use a client-side cursor (discussed later in this chapter).
`Source`	A `Variant` which can either be a `String` containing the name of a table, a SQL Statement, or the name of a stored procedure or an object reference to a `Command` object.
`State`	A `Long` value describing the current state of the `Recordset` object. Multiple values can be combined to describe the current state (see Table 14-7 for a list of values).
`Status`	A `Long` value describing the current status of a record in a batch update (see Table 14-8).
`StayInSync`	A `Boolean` value that when `TRUE` means that the reference to child records in a hierarchical `Recordset` will automatically be changed when the parent row's position changes. The default value for this property is `TRUE`.

Table 14-2
Values for CursorLocation

Constant	*Value*	*Description*
`adUseNone`	1	Doesn't use cursor services. This value should not be selected and exists solely for compatibility.
`adUseServer`	2	The cursor services are provided by the data provider.
`adUseClient`	3	The cursor services are provided by a local cursor library, which may provide more features than the data provider's cursor library provides.

Table 14-3
Values for CursorType

Constant	*Value*	*Description*
`adOpenUnspecified`	-1	The type of cursor isn't specified.
`adOpenForwardOnly`	0	A forward-only cursor is used, which permits you only to scroll forward through the records in the `Recordset`.
`adOpenKeyset`	1	A keyset cursor is used, which is similar to a dynamic cursor, but doesn't permit you to see records added by other users.
`adOpenDynamic`	2	A dynamic cursor is used, which allows you to see records added by other users, plus any changes and deletions made by other users.
`adOpenStatic`	3	A static cursor is used, which prevents you from seeing any and all changes from other users.

Table 14-4
Values for EditMode

Constant	***Value***	***Description***
`adEditNone`	0	Editing is not active; the data in the current record hasn't been changed.
`adEditInProgress`	1	The data in the current record has been changed, but not yet saved.
`adEditAdd`	2	The `AddNew` method has been used to create a new record, but the record hasn't been saved yet.
`adEditDelete`	4	The current record has been deleted.

Table 14-5
Values for Filter

Constant	***Value***	***Description***
`adFilterNone`	0	Removes all filtering criteria.
`adFilterPendingRecords`	1	Selects only those records that have been changed, but not sent to the server for updating. Applies to batch update mode only.
`adFilterAffectedRecords`	2	Selects only those records that have been affected by the last `CancelBatch`, `Delete`, `Resync`, or `UpdateBatch`.
`adFilterFetchedRecords`	3	Selects only those records in the current cache.
`adFilterConflictingRecords`	5	Selects only those records that have failed the last batch update.

Table 14-6
Values for LockMode

Constant	***Value***	***Description***
`adLockUnspecified`	-1	The type of locking isn't specified.
`adLockReadOnly`	1	Doesn't permit you to change any values.

Constant	***Value***	***Description***
`adLockPessimistic`	2	Records are locked at the data source record by record once the data in the record has been changed.
`adLockOptimistic`	3	Records are locked only when you call the `UpdateMethod`.
`adLockBatchOptimistic`	4	Records are not locked, and conflicts will be returned for resolution after the `UpdateBatch` method has completed.

Table 14-7
Values for State

Constant	***Value***	***Description***
`adStateClosed`	0	The `Command` object is closed.
`adStateOpen`	1	The `Command` object is open.
`adStateConnecting`	2	The `Command` object is connecting to the database.
`adStateExecuting`	4	The `Command` object is executing.
`adStateFetching`	8	Rows are being retrieved.

Table 14-8
Values for Status

Constant	***Value***	***Description***
`adRecOK`	0	The record was successfully updated.
`adRecNew`	1	The record is new.
`adRecModified`	2	The record has been modified.
`adRecDeleted`	4	The record has been deleted.
`adRecUnmodified`	8	The record has not been modified.
`adRecInvalid`	16	The record wasn't saved because its bookmark was invalid.

Continued

Table 14-8 *(continued)*

Constant	***Value***	***Description***
`adRecMultipleChanges`	64	The record wasn't saved because it would have affected multiple records.
`adRecPendingChanges`	128	The record wasn't saved because it refers to a pending insert.
`adRecCanceled`	256	The record wasn't saved because the operation was canceled.
`adRecCantRelease`	1024	The new record wasn't saved because the existing record was locked.
`adRecConcurrencyViolation`	2048	The record wasn't saved because optimistic concurrency was in use.
`adRecIntegrityViolation`	4096	The record wasn't saved because the data violated an integrity constraint.
`adRecMaxChangesExceeded`	8192	The record wasn't saved because there were too many pending changes.
`adRecObjectOpen`	16384	The record wasn't saved because of a conflict with an open storage object.
`adRecOutOfMemory`	32768	The record wasn't saved because the computer ran out of memory.
`adRecPermissionDenied`	65536	The record wasn't saved because the user doesn't have sufficient permissions.
`adRecSchemaViolation`	131072	The record wasn't saved because it violates the underlying database structure.
`adRecDBDeleted`	262144	The record has already been deleted from the data source.

Recordset object methods,

The `Recordset` object contains many different methods to manipulate the data in the `Recordset`.

Sub AddNew ([FieldList], [Values])

The `AddNew` method is used to add a new empty record to the `Recordset`. This will set the `EditMode` property to `adEditAdd`. After assigning values to each of the fields, you can use the `Update` method to save the changes. If you specify the `FieldList` and `Values` parameters, the values are immediately saved to the database and no call to `Update` is needed.

In batch update mode, you proceed as above. The changes will be saved locally in the Recordset, but not sent to the server. After adding the last record of the batch, you must call the UpdateBatch method to save the changes to the database. Then you should set the Filter property to adFilterConflicting and take the appropriate action for the records that weren't posted to the database properly.

FieldList is optional and is either a String value containing a single field name or a String Array containing a list of field names.

Values is an optional Variant containing a single value corresponding to a single field name or Array of values corresponding to the Array of field names.

Sub Cancel()

The Cancel method is used to terminate an asynchronous task started by the Open method.

Sub CancelBatch ([AffectRecords As AffectEnum = adAffectAll])

The CancelBatch method is used to cancel a pending batch update. If the current record hasn't been saved, CancelBatch will automatically call CancelUpdate to discard any changes. As with all batch operations, you should verify the Status property for all of the affected records to ensure that they were properly canceled.

AffectRecords is an enumerated type that describes which records will be affected by the cancel operation (see Table 14-9).

Table 14-9
Values for AffectRecords

Constant	*Value*	*Description*
adAffectCurrent	1	Affects only the current record in the Recordset.
adAffectGroup	2	Affects only those records selected by the current value of the Filter property.
adAffectAll	3	Affects all records in the Recordset.
adAffectAllChapters	4	Affects all records in all chapters of the Recordset.

Sub CancelUpdate ()

The CancelUpdate method is used to abandon all of the changes made to a record and to restore its values to the original values before the Update method is called. If you use CancelUpdate to undo a row added with the AddNew method, the newly added row will be discarded, and the current record will become the row that was the current row before the AddNew method was used.

Function Clone ([LockType As LockTypeEnum = adLockTypeUnspecified]) As Recordset

The Clone method is used to create a duplicate of a Recordset object. LockType allows you to specify that the new Recordset object is read-only. The only permissible values are adLockUnspecified and adLockReadOnly.

Sub Close

The Close method closes an open Recordset and frees all of its associated resources. If you are editing a record in the Recordset, an error will occur, and the Close method will be terminated. You need to call either Update or CancelUpdate first. If you are working in batch mode, all changes since the last UpdateBatch will be lost.

Sub CompareBookmarks(Bookmark1, Bookmark2) As CompareEnum

The CompareBookmarks method compares two bookmarks and returns information about their relative positions. Both bookmarks must come from the same Recordset object or from clones of the same Recordset object. Table 14-10 contains a list of possible return values.

Bookmark1 and Bookmark2 are valid bookmarks.

Table 14-10
Values for CompareEnum

Constant	*Value*	*Description*
adCompareLessThan	0	The first bookmark is before the second bookmark.
adCompareEqual	1	The two bookmarks are equal.
adCompareGreaterThan	2	The first bookmark is after the second bookmark.
adCompareNotEqual	3	The two bookmarks are different and not ordered.
adCompareNotComparable	4	The bookmarks can't be compared.

Sub Delete ([AffectRecords As AffectEnum = adAffectCurrent])

The Delete method is used to delete one or more records from a Recordset object.

AffectRecords is an enumerated type that describes which records will be deleted by this operation (see Table 14-9). The default value is adAffectCurrent, which means only the current row will be deleted.

Sub Find (Criteria As String, [SkipRows As Long], [SearchDirection As SearchDirectionEnum = adSearchForward], [Start])

The Find method is used to locate the specified criteria in a Recordset. Note that you must have a valid current record before calling the Find method, so you may want to call MoveFirst before using this method.

Criteria is a String value that specifies a field name, a comparison operator, and a value. Only one field name may be specified.

SkipRows is a Long value that specifies the number of rows to skip relative to the current row before starting the search. The default value is zero, which means the search will begin with the current row.

SearchDirection is an enumerated type that indicates the direction of the search. You can specify adSearchForward (1) to search to the end of the Recordset, or you can specify adSearchBackward (-1) to search to the beginning of the Recordset.

Start is a Variant containing a bookmark for the first record to be searched.

Function GetRows ([Rows As Long = -1], [Start], [Fields]) As Variant

The GetRows method retrieves multiple records from a Recordset object into a two-dimensional array.

Rows is the number of rows to be retrieved. The default value is adGetRowsRest (-1), which will retrieve the rest of the rows in the table.

Start is an optional Variant containing a bookmark for the first row to be retrieved. If you do not specify values for both Rows and Start, all of the rows in the table will be retrieved.

Fields is an optional Variant containing a String with a single field name, a String array with multiple field names, the index of the field, or an array of field index values.

Function GetString ([StringFormat As StringFormatEnum = adClipString], [NumRows As Long = -1], [ColumnDelimiter As String], [RowDelimiter As String], [NullExpr As String]) As String

The `GetString` method returns a `String` containing the values from the `Recordset`.

`StringFormat` is a `Long` value containing the value `adClipString` (2). This is the only legal value for this method.

`NumRows` is a `Long` value containing the number of rows to be saved in the string. A value of –1 means that all of the rows will be retrieved.

`ColumnDelimiter` is a `String` containing the delimiter to be used between each column. If not specified, a tab character will be used.

`RowDelimiter` is a `String` containing the delimiter to be used between each row. If not specified, a carriage return character will be used.

`NullExpr` is a `String` containing the value to be displayed in place of a null value. If it is not specified, nothing will be output.

Tip

To CSV or not to CSV: I often find it useful to create Comma Separated Value files from a database. These files can be easily imported into a program like Excel for analysis and testing. The `GetString` method makes it easy to write a program to save your data into a CSV file. Simply specify a comma for `ColumnDelimiter` and `vbCrLf` for the `RowDelimiter`.

Sub Move (NumRecords As Long, [Start])

The `Move` method is used to move the current record pointer to a different location in the `Recordset`. If the `Recordset` is already at `BOF`, an attempt to move backwards will generate a runtime error. Similarly, if the `Recordset` is already at `EOF`, an attempt to move forward will also generate a runtime error.

`NumRecords` is a `Long` value containing the number of records to be moved. If a value of zero is specified, the current record pointer remains unchanged, and the current record is refreshed. A value greater than zero means that the current record position will be moved to the end of the `Recordset`, while a value less than zero will move the current record pointer towards the beginning of the `Recordset`.

`Start` is a `Variant` containing a bookmark that will be used as the starting position for the move.

Tip

Forward-NOT-only: Even if you specify a forward-only cursor, you can still move backward using the `Move` method, as long as you do not move beyond the records in the current cache.

Sub MoveFirst ()

The MoveFirst method moves the current record pointer to the first record in the Recordset.

Tip

MoveFirst first: I usually find it a good idea to call MoveFirst after opening a Recordset to ensure that I have a valid record ready for processing.

Sub MoveLast()

The MoveLast method moves the current record pointer to the last record in the Recordset. Note that you can't use the MoveLast method with a forward-only cursor.

Sub MoveNext ()

The MoveNext method moves the current record pointer to the next record in the Recordset. If the current record pointer is at the last record in the Recordset, the current record pointer will be moved to EOF. If the current record pointer is already at EOF, a call to MoveNext will cause a runtime error.

Sub MovePrevious ()

The MovePrevious method moves the current record pointer to the previous record in the Recordset. If the current record pointer is on the first record in the Recordset, the current record pointer will be moved to BOF. If the current record pointer is already at BOF, a call to MovePrevious will trigger a runtime error.

Function NextRecordset ([RecordsAffected]) As Recordset

The NextRecordset method clears the current Recordset object and returns the next Recordset that resulted from a query or stored procedure that returned multiple Recordsets. RowsAffected is a Long value containing the number of records affected.

Sub Open ([Source As Variant], [ActiveConnection As Variant], [CursorType As CursorTypeEnum = adOpenUnspecified], [LockType As LockTypeEnum], [Options As Long = -1])

The Open method opens a new Recordset object.

Source is a Variant containing an object reference to a valid Command object; an object reference of a valid Stream object containing a persistently stored Recordset; or a String containing a SQL statement, a table name, the name of a stored procedure, a URL, or the name of a file.

`ActiveConnection` is a `Variant` that contains an object reference to an open `Connection` object or a `String` value that contains the same connection information found in the `ConnectionString` property.

`CursorType` is an enumerated value (see Table 14-11) that specifies the type of cursor that will be used on the `Recordset`.

`LockType` is an enumerated value (see Table 14-12) containing the locking mode used by the `Recordset` when retrieving records.

`Options` optionally passes one of the values specified in Table 14-3.

Table 14-11
Values for CursorType

Constant	*Value*	*Description*
`adOpenUnspecified`	-1	The type of cursor isn't specified.
`adOpenForwardOnly`	0	A forward-only cursor is used, which permits you only to scroll forward through the records in the `Recordset`.
`adOpenKeyset`	1	A keyset cursor is used, which is similar to a dynamic cursor, but doesn't permit you to see records added by other users.
`adOpenDynamic`	2	A dynamic cursor is used, which allows you to see records added by other users, plus any changes and deletions made by other users.
`adOpenStatic`	3	A static cursor is used, which prevents you from seeing any and all changes from other users.

Table 14-12
Values for LockType

Constant	*Value*	*Description*
`adLockUnspecified`	-1	The type of locking isn't specified.
`adLockReadOnly`	1	Doesn't permit you to change any values.
`adLockPessimistic`	2	Records are locked at the data source record by record once the data in the record has been changed.

Constant	*Value*	*Description*
adLockOptimistic	3	Records are locked only when you call the UpdateMethod.
adLockBatchOptimistic	4	Records are not locked, and conflicts will be returned for resolution after the UpdateBatch method has completed.

Sub Requery ([Options As Long = -1])

The Requery method gets a fresh copy of the data in a Recordset by re-executing the query that originally generated the Recordset.

Options is a Long value that describes how to execute the query. You may set its value to any combination of the following values described in Table 14-13: adAsyncExecute, adAsyncFetch, adAsynchFetchNonBlocking, and adExecuteNoRecords. If omitted, none of these values will be selected.

Table 14-13
Values for Options

Constant	*Value*	*Description*
adOptionUnspecified	-1	No options are specified.
adCmdText	1	CommandText contains either a SQL statement or a stored procedure call.
adCmdTable	2	CommandText contains the name of a table in the database.
adCmdStoredProcedure	4	CommandText contains the name of a stored procedure.
adCmdUnknown	8	The type of command isn't known.
adAsyncExecute	16	The command should be executed asynchronously.
adAsyncFetch	32	After the number of rows specified in the CacheSize property of the Recordset object are returned, the remaining rows will be returned asynchronously.

Continued

Table 14-13 *(continued)*

Constant	***Value***	***Description***
adAsyncFetchNonBlocking	64	The main thread isn't blocked while retrieving rows. If the current row hasn't been retrieved, the current row will be moved to the end of the file.
adExecuteNoRecords	128	Indicates that the command will not return any rows or will automatically discard any rows that are generated. Must be used with either the adCmdText or adCmdStoredProcedure values.
adCmdFile	256	CommandText is the name of a persistently stored Recordset.
adCmdTableDirect	512	CommandText contains the name of a database table.

Sub Resync ([AffectRecords As AffectEnum], [ResyncValues As ResyncEnum)

The Resync method gets any updates to the data that may have happened while using a static or forward-only cursor. Unlike the Requery method, the Resync method does not re-execute the query associated with the Recordset, so any rows added since the original query was executed will not be visible.

AffectRecords is an enumerated type that describes which records will be affected by the cancel operation (see Table 14-9).

ResyncValues is an enumerated type that determines how changed records in the Recordset will be handled. A value of adResyncUnderlyingValues (1) means that all pending updates are saved, while a value of adResyncAllValues (2) means that all pending updates are canceled (default).

Sub Save ([Destination], [PersistFormat As PersistFormatEnum])

The Save method is used to save a local copy of an open Recordset (including any child recordsets associated with it) to a disk file or a Stream object. If you have an active filter on the Recordset, only the filtered records will be saved. After the first time you call the Save method, you should omit the Destination parameter, because the destination remains open. If you specify the same value for Destination a second time, a runtime error will occur. If you specify a different value, both destinations will remain open. Closing the Recordset will also close the Destination.

`Destination` is a `Variant` value containing either an object reference to a `Stream` or a `String` containing the fully qualified filename where the data will be stored.

`PersistFormat` is an enumerated data type whose value is either `adPersistADTG` (0), which will save the `Recordset` in the Advanced Data TableGram (ADTG) format (default), or `adPersistXML` (1), which will save the `Recordset` using XML format.

Sub Seek (KeyValues, [SeekOption As SeekEnum])

The `Seek` method is used to move the cursor to a new location in the `Recordset`. It works with the `Index` property to search the specified index for a particular value.

`KeyValues` is an array of `Variant` values, corresponding to the columns in the index.

`SeekOption` is an enumerated data type that specifies how to perform the comparison (see Table 14-14).

Table 14-14
Values for SeekOption

Constant	*Value*	*Description*
`adSeekFirstEQ`	1	Seeks the first row with a key value equal to `KeyValues`.
`adSeekLastEQ`	2	Seeks the last row with a key value equal to `KeyValues`.
`adSeekAfterEQ`	4	Seeks the first row with a key value greater than or equal to `KeyValues`.
`adSeekAfter`	8	Seeks the first row with a key value greater than `KeyValues`.
`adSeekBeforeEQ`	16	Seeks the first row with a key value just less than or equal to `KeyValues`.
`adSeekBefore`	32	Seeks the first row with a key value just less than `KeyValues`.

Note

Clients not wanted: The `Seek` method will not work with client-side cursors (discussed later in this chapter), so select `adUseServer` for the `CursorType` property.

Function Supports (CursorOptions As CursorOptionEnum) As Boolean

The Supports method returns TRUE if the data provider supports the combination of values specified in CursorOptions. CursorOptions is a Long value whose value is created by adding one or more values of the values listed in Table 14-15 together.

Table 14-15
Values for CursorOptions

Constant	*Value*	*Description*
adHoldRecords	256	The provider will retrieve more records or changes to another position without committing all pending changes.
adMovePrevious	512	The provider supports the MoveFirst, MovePrevious, Move, and GetRows methods to move the cursor backwards without requiring bookmarks.
adBookmark	8192	The provider supports the Bookmark property.
adApproxPosition	16384	The provider supports the AbsolutePosition and AbsolutePage properties.
adUpdateBatch	65536	The provider supports the UpdateBatch and Cancel batch methods.
adResync	131072	The provider supports the Resync method.
adNotify	262144	The provider supports notifications, which implies that Recordset events are supported.
adFind	524288	The provider supports the Find method.
adSeek	4194304	The provider supports the Seek method.
adIndex	8388608	The provider supports the Index property.
adAddNew	16778240	The provider supports the AddNew method.
adDelete	16779264	The provider supports the Delete position.
adUpdate	16779984	The provider supports the Update method.

Sub Update ([Fields], [Values])

The Update method saves any changes you make the current row of a Recordset. You can either update the row directly by accessing each field by using the Recordset's Fields collection, or you can specify a list of fields and their values by using the Fields and Values properties.

If you move to another row in the Recordset, ADO will automatically call the Update method to save your changes. You must explicitly call the CancelUpdate method to discard any changes you may have made before moving to another row.

Fields is a Variant value containing a single field name or a Variant array containing a list of field names.

Values is a Variant value containing a single value or a Variant array containing a list of values, whose position in the array correspond to the field names in the Fields parameter.

Note

All or none: You must specify both the Fields and Values parameters or neither. Specifying only one will cause an error. Also if you pass an array for Fields, the Values property must also be an array of an identical size.

Sub UpdateBatch ([AffectRecords As AffectEnum])

The UpdateBatch method saves all pending batch changes to the database. AffectRecords is an enumerated data type indicating which rows in the Recordset will be affected (see Table 14-9).

Caution

It sort of worked: You must check the Errors collection after performing a UpdateBatch to determine which rows weren't properly updated. You can set the Filter property to adFilterAffectedRecords and move though the remaining records to identify those with problems by checking the Status property.

Recordset object events

The Recordset object contains many different events to handle various conditions encountered in the Recordset.

Event EndOfRecordset (fMoreData As Boolean, adStatus As EventStatusEnum, pRecordset As Recordset)

The EndOfRecordset event is triggered when the program attempts to move beyond the end of the Recordset. This is most likely triggered by a call to MoveNext. You can use this event to acquire more rows from the database and append them to the end of Recordset to allow the MoveNext to succeed. If you do this, you must set the fMoreData parameter to TRUE to indicate that the cursor is no longer at the end of the Recordset.

fMoreData is a Boolean value where TRUE means that more rows have been added to the Recordset.

adStatus is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16). When the event is triggered, this value will be set to either adStatusOk or adStatusCantDeny. If it is set to adStatusCantDeny, you can't set the parameter to adStatusCancel.

Table 14-16
Values for EventStatusEnum

Constant	***Value***	***Description***
AdStatusOK	1	The operation that triggered the event was successful.
adStatusErrorsOccured	2	The operation that triggered the event failed.
adStatusCantDeny	3	The operation can't be canceled.
adStatusCancel	4	Requests that the operation that triggered the event be canceled.
adStatusUnwantedEvent	5	Prevents subsequent notifications before the event method has finished executing.

pRecordset is an object reference to the Recordset that triggered the event.

Event FetchComplete (pError As Error, adStatus As EventStatusEnum, pRecordset As Recordset)

The FetchComplete event is triggered after all of the rows have been retrieved during an asynchronous operation.

pError is an object reference to an Error object containing any error information. This property is only valid if adStatus is set to adStatusErrorsOccured.

adStatus is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16). You may also set this value to adStatusUnwantedEvent before the event completes to prevent it from being called again.

pRecordset is an object reference to the Recordset that triggered the event.

Event FetchProgress (Progress As Long, MaxProgress As Long, adStatus As EventStatusEnum, pRecordset As Recordset)

The FetchProgress event is fired periodically during an asynchronous operation to report how many rows have been retrieved.

Progress is a Long value indicating the number of records that have been retrieved so far.

MaxProgress is a Long value indicating the number of records that are expected to be retrieved.

adStatus is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

pRecordset is an object reference to the Recordset that triggered the event.

Event FieldChangeComplete (cFields As Long, Fields, pError asError, adStatus As EventStatusEnum, pRecordset As Recordset)

The FieldChangeComplete event is called after the values of one or more Field objects have been changed. This can happen if the program assigns a value to the Value property of the Field object or by using the Update method and specifying a list of fields and values.

cFields is a Long value indicating the number of fields in Fields.

Fields is an array of object pointers that point to Field objects that were changed.

pError is an Error object containing any error that may have occurred. This value is valid only if adStatus is set to adStatusErrorsOccured.

adStatus is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

pRecordset is an object reference to the Recordset that triggered the event.

Event MoveComplete (adReason As EventReasonEnum, pError As Error, adStatus As EventStatusEnum, pRecordset as Recordset)

The MoveComplete event is called after the current record has changed to a new position in the Recordset.

Tip

Counting rows: If you want to display the relative position of the cursor in the Recordset, the MoveComplete event is an ideal place for this. You can easily use the AbsolutePosition property to display the current record number and the RecordCount properties to display the total number of records retrieved.

adReason is an enumerated data type indicating the operation that originally caused the move (see Table 14-17).

Table 14-17
Values for EventReasonEnum

Constant	*Value*	*Description*
`adRsnAddNew`	1	The operation executed an `AddNew` method.
`adRsnDelete`	2	The operation used the `Delete` method.
`adRsnUpdate`	3	The operation performed an `Update` method.
`adRsnUndoUpdate`	4	The operation reversed an `Update` method.
`adRsnUndoAddNew`	5	The operation reversed an `AddNew` method.
`adRsnUndoDelete`	6	The operation reversed a `Delete` method.
`adRsnRequery`	7	The operation performed a `Requery`.
`adRsnResync`	8	The operation performed a `Resync`.
`adRsnClose`	9	The operation closed the `Recordset`.
`adRsnMove`	10	The operation moved the current record pointer to a different record.
`adRsnFirstChange`	11	The operation made the first change to a row.
`adRsnMoveFirst`	12	The operation moved the current record pointer to the first record in the `Recordset`.
`adRsnMoveNext`	13	The operation moved the current record pointer to the next record in the `Recordset`.
`adRsnMovePrevious`	14	The operation moved the current record pointer to the previous record in the `Recordset`.
`adRsnMoveLast`	15	The operation moved the current record pointer to the last record in the `Recordset`.

`pError` is an `Error` object containing any error that may have occurred. This value is only valid if `adStatus` is set to `adStatusErrorsOccured`.

`adStatus` is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

`pRecordset` is an object reference to the `Recordset` that triggered the event.

Event RecordChangeComplete (adReason As EventReasonEnum, cRecords As Long, pError As Error, adStatus As EventStatusEnum, pRecordset as Recordset)

The RecordChangeComplete event is called after one or more records in the Recordset have been changed.

adReason is an enumerated data type indicating the operation that originally caused the change (see Table 14-17). Possible values are the following: adRsnAddNew, adRsnDelete, adRsnUpdate, adRsnUndoUpdate, adRsnUndoAddNew, adRsnUndoDelete, and adRsnFirstChange.

cRecords is a Long value containing the number of records that were changed.

pError is an Error object containing any error that may have occurred. This value is only valid if adStatus is set to adStatusErrorsOccured.

adStatus is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

pRecordset is an object reference to the Recordset that triggered the event.

Event RecordsetChangeComplete (adReason As EventReasonEnum, pError As Error, adStatus As EventStatusEnum, pRecordset as Recordset)

The RecordsetChangeComplete event is called after a change to the Recordset.

adReason is an enumerated data type indicating the operation that originally caused the move (see Table 14-17). Possible values are the following: adRsnClose, adRsnReQuery, and adRsnReSync.

pError is an Error object containing any error that may have occurred. This value is only valid if adStatus is set to adStatusErrorsOccured.

adStatus is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

pRecordset is an object reference to the Recordset that triggered the event.

Event WillChangeField (cFields As Long, Fields, adStatus As EventStatusEnum, pRecordset As Recordset)

The `WillChangeField` event is called before an operation that will change the values in one or more `Field` objects is started. After the changes have been made, the `FieldChangeComplete` event is fired. You can choose to cancel the operation by setting the `adStatus` property to `adStatusCancel`.

`cFields` is a `Long` value indicating the number of fields in `Fields`.

`Fields` is an array of object pointers that point to `Field` objects that are to be changed.

`adStatus` is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

`pRecordset` is an object reference to the `Recordset` that triggered the event.

Event WillChangeRecord (adReason As EventReasonEnum, cRecords As Long, pError As Error, adStatus As EventStatusEnum, pRecordset as Recordset)

The `WillChangeRecord` event is called before one or more records in the `Recordset` will be changed.

`adReason` is an enumerated data type indicating the operation that caused the change (see Table 14-17). Possible values are the following: `adRsnAddNew`, `adRsnDelete`, `adRsnUpdate`, `adRsnUndoUpdate`, `adRsnUndoAddNew`, `adRsnUndoDelete`, and `adRsnFirstChange`.

`cRecords` is a `Long` value containing the number of records will be changed.

`adStatus` is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

`pRecordset` is an object reference to the `Recordset` that triggered the event.

Event WillChangeRecordset (adReason As EventReasonEnum, pError As Error, adStatus As EventStatusEnum, pRecordset as Recordset)

The `WillChangeRecordset` event is called before the `Recordset` is changed.

`adReason` is an enumerated data type indicating the operation that caused the change (see Table 14-17). Possible values are the following: `adRsnClose`, `adRsnReQuery`, and `adRsnReSync`.

`pError` is an `Error` object containing any error that may have occurred. This value is only valid if `adStatus` is set to `adStatusErrorsOccured`.

`adStatus` is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

`pRecordset` is an object reference to the `Recordset` that triggered the event.

Event WillMove (adReason As EventReasonEnum, adStatus As EventStatusEnum, pRecordset as Recordset)

The `WillMove` event is called before the current record pointer is changed to a new position in the `Recordset`.

`adReason` is an enumerated data type indicating the operation that originally caused the move (see Table 14-16).

`adStatus` is an enumerated value that indicates the action that should be taken when the event returns (see Table 14-16).

`pRecordset` is an object reference to the `Recordset` that triggered the event.

Before Opening a Recordset

Before you open a recordset there are a number of issues you need to consider. For instance, you need to decide whether or not you plan to update the database. Then you need to decide what type of cursor you want to use. Finally, you need to decide where the cursor should be located. These issues have a big impact on how your application performs, not only for a single user, but for all of the users that access your database server.

Locking considerations

One of the problems of running an application where multiple users are accessing the same collection of information is controlling access to the information so that two people aren't trying to update the same information at the same time.

To understand why locking is important consider the following example. Assume that your bank runs two programs to update their accounts at the same time (see Figure 14-1). One program applies deposits, while the other applies withdrawals. As luck would have it, both programs attempt to update your account at the same time. Without locking, the withdrawal program may read the current balance in your account and begin to process the withdrawal. Then the deposit program reads the current balance and updates the database with the new balance. Finally, the withdrawal program posts the balance it computes. This is a serious problem (although I wish it worked the other way).

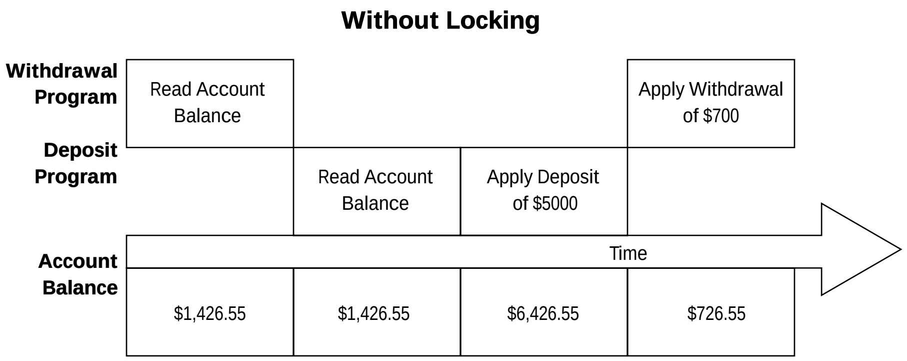

Figure 14-1: Processing concurrent database updates without locks

To avoid this problem, you need to prevent all other users from accessing this particular account until after the update has been completed. Figure 14-2 shows how the same sequence of actions would work with locks. The withdrawal program is able to read the account balance immediately, and it places a lock on the account so no other programs can access the information. When the deposit program attempts to read the balance, it is forced to wait for the lock to be released. This allows the withdrawal program to finish its processing uninterrupted and release the lock it placed on the account. Once the lock is released, the account balance will be returned to the deposit program, which can then complete its updates accurately.

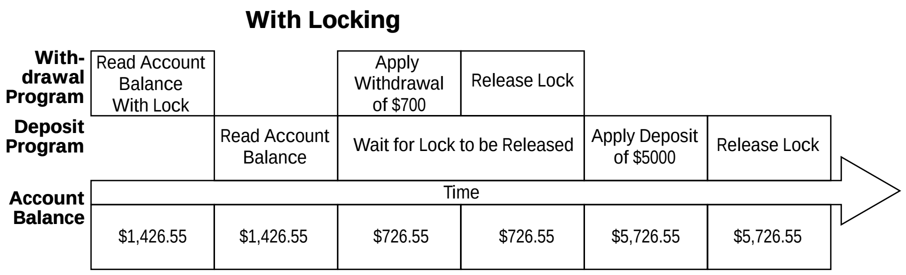

Figure 14-2: Processing concurrent database updates with locks

As you can see, locking is necessary to ensure that changes to the database are applied in the correct order. In most cases, the order of the changes against the database isn't important as long as the changes are made sequentially rather than concurrently. After all, it doesn't matter whether the deposit is made first or the withdrawal is made first as long as they aren't made at the same time.

The ADO `LockType` allows you to choose from one of four different types of locks: read-only, pessimistic, optimistic, and batch optimistic. You need to choose the one that best suits your needs.

Read-only locks

While locks are necessary when performing updates, it is useful to tell the database server that you are never going to update the data in the database. In a sense, a *read-only lock* is not a lock at all, but specifying a read-only lock ensures that you can't update the database.

Pessimistic locks

A *pessimistic lock* is the most conservative lock available in ADO. When you begin to edit a record by assigning a new value into one of the existing fields in the row, a lock is placed on the row so that no other users can access it. The row remains locked until you update it using the `Update` method.

One problem with using pessimistic locking is that in addition to the row you are modifying, all of the rows stored in the same physical block of storage (typically called a *database page*) are also locked, which makes them unavailable for other users also.

Note

In days of old: The current version of most popular database systems supports row-level locking, which means that a lock affects only the row that it was intended for and not the other rows in the same physical block of storage.

However, a bigger problem with pessimistic locking is that the lock is held during the entire time you are editing the row. Depending on how your application works, the lock could be held anywhere from a few moments to many minutes. In some applications this may not be critical, but in others it could cause serious problems because other applications may be forced to wait until you complete the edit. Also the database server may take the lock away from you if you held it too long, which will cause an error when you eventually get around to finishing the update.

In general, the more rows you have in a table, the fewer problems you will have with conflicting pessimistic locks. Likewise, performing fewer updates and having fewer users also reduces the likelihood of a lock conflict.

Optimistic locks

Optimistic locks still ensure that your update is performed correctly. Unlike pessimistic locks, however, optimistic locks aren't placed until you call the `Update` method, so a lock is not placed on the row when you begin editing the row. When the update is performed, the row is locked, and the current value of each field in the database server is compared to the original value of each field taken when the application program retrieved the row. If there aren't any differences, the update will proceed normally. Otherwise an error will be returned to your program. It will be up to your program to decide whether to reapply the changes or restart the update from the beginning.

Batch optimistic locks

Batch optimistic locks are similar to optimistic locks, except you can update multiple rows before returning them to the database server for updating using the `UpdateBatch` method. After performing a batch update, you need to review each of the rows you updated to ensure that the changes were applied to the database.

This type of lock is useful when you want to add lots of rows to a table. It is impossible for anyone to update any of the values in these rows, since the data hasn't been sent to the database yet.

Choosing a cursor type

While locks affect how you update data, cursor types affect how your application will see changes in the database that are made by other users. Choosing the appropriate cursor type is important and depends mostly on how you plan to use the data. ADO supports four types of cursors: forward-only cursors, static cursors, keyset cursors, and dynamic cursors.

Forward-only cursors

A *forward-only cursor* is the most restrictive of all of the cursors available. It is available only in combination with a read-only lock. It presents a static view of the data in the `Recordset` that can't be addressed randomly. Any changes, additions, or deletions to the underlying data will not be visible to your program.

You must scroll through the `Recordset` one row at a time from the beginning to the end. If you need to move backwards or start over again, you will have to close the `Recordset` and reopen it.

Typically this type of cursor is most useful for reports or for translation tables that are never updated. Forward-only cursors have less overhead and offer better performance than the other types of cursors.

Static cursors

Like the forward-only cursor, a *static cursor* also provides a static view of your data. Records that have been added, deleted, or updated will not be visible to your program. However, you may move the current record pointer to any location in the `Recordset` without restriction.

Note **Client-side only:** A static cursor is your only choice for client-side cursors, though it may be used with a server-side cursor as well.

Keyset cursors

A *keyset cursor* allows you to see any updates or deletions, but not additions made to the underlying data that have been made after you opened the `Recordset`. Bookmarks are supported, and you may move anywhere in the `Recordset` you choose.

Keyset cursors work by keeping a list of bookmarks in a temporary table on the server. As you move from one row to another, the provider will retrieve the most current values associated with the particular row. In general, I avoid keyset cursors because they are slower than static cursors and don't provide all of the updates to the database as does a dynamic cursor.

Dynamic cursors

Dynamic cursors allow you to view any changes made to the database, including additions, deletions, and updates. All forms of movement through the `Recordset` that don't rely on `Bookmarks` are always supported. Note that most data providers include bookmark support even though it isn't required.

Dynamic cursors are useful if you want to see all of the changes in the underlying data you retrieved from the database. They are only appropriate if you are using a server-side cursor and there is more overhead when using a dynamic cursor because the provider has to check continually for changes in the database. However, if you are only retrieving a few rows from the database, then dynamic cursors may be appropriate.

Picking a cursor location

The last major choice you have to make when opening a recordset is choosing between a server-side cursor and a client-side cursor. This choice is made using the `CursorLocation` property.

Server-side cursors

A *server-side cursor* is the traditional cursor used in a database system. It directly accesses data on the server. A request to move the current record pointer to a different record results in a request to the database server (unless, of course, you set `CacheSize` to a value greater than one, and the record is in the local cache). Server-side cursors can be used with keyset and dynamic cursors to allow your program to see changes in the live database.

Client-side cursors

A *client-side cursor* is a special type of cursor that allows you a richer environment with which to build your application than a server-side cursor does. Data is buffered locally, and you can operate in disconnected mode from the database server. After retrieving your data from the database, you can break the connection and work with the information in the recordset locally. Later you can reconnect and apply whatever changes you made locally to the database server.

Note

I can't see it: The `UnderlyingValue` property is not available on `Field` objects using a client-side cursor, since all of the records are buffered locally.

Opening a Recordset

There are three main ways to create a `Recordset`. As you saw in Chapter 13, you can create a `Command` object and use the `Execute` method to return a reference to a `Recordset` object. A second way to create a `Recordset` is by storing a SQL statement, stored procedure name, or table name in the recordset's `Source` property and then using the `Open` method to populate the recordset. The last way is to set the `Source` property to an active `Command` object and then use the `Open` method. Because I already covered the first way in Chapter 13, I'll cover the last two ways here.

Using Source strings

For many applications all you need to use is a `Connection` object and a `Recordset` object. Listing 14-1 is similar to the program in Chapters 13 and uses the same `Connection` object from Chapter 12. Instead of creating a `Command` object with the **Select** statement, I assign the **Select** statement to the `Source` property of the `Recordset`.

I then specify values for `CursorLocation`, `CursorType`, and `LockType`. While these values aren't critical to the way this routine runs, it is a good idea to specify them always and ask only for the resources you really need. Then after clearing the

Errors collection, I issue the Open method. If there's a problem, I write the error message using the WriteError routine from Chapter 12. Otherwise, I display the first value from the first column of the recordset in the status bar just like I did in the previous version of the program.

Listing 14-1: **The Command7_Click event in Recordset Demo**

```
Private Sub Command7_Click()

Dim rs As ADODB.Recordset

On Error Resume Next

Set rs = New ADODB.Recordset
Set rs.ActiveConnection = db
rs.Source = "Select Count(*) From Customers Where State = 'MD'"
rs.CursorLocation = adUseServer
rs.CursorType = adOpenForwardOnly
rs.LockType = adLockReadOnly

db.Errors.Clear
rs.Open
If db.Errors.Count > 0 Then
   WriteError

Else
   StatusBar1.SimpleText = "Response: " & _
      FormatNumber(rs.Fields(0).Value, 0)

End If

End Sub
```

Using Command objects

Another way to populate a Recordset object is to create a Command object with the command to extract the information from the database and assign it to the record-set's Source property. Listing 14-2 is essentially a combination of the code from Listing 14-1 and Listing 13-5. It takes the steps I used to build a Command object that references the stored procedure CountByState and marries it to the code to create and open the Recordset. The only difference in the steps is that I used a Set statement to assign the Command object to the recordset's Source property instead of assigning a **Select** statement to the Source property.

Listing 14-2: **The Command8_Click event in Recordset Demo**

```
Private Sub Command8_Click()

Dim c As ADODB.Command
Dim p As ADODB.Parameter
Dim rs As ADODB.Recordset

On Error Resume Next

Set c = New ADODB.Command
Set c.ActiveConnection = db
c.CommandText = "CountByState"
c.CommandType = adCmdStoredProc

c.Parameters.Refresh
c.Parameters("@State").Value = Text5.Text

Set rs = New ADODB.Recordset
Set rs.Source = c
rs.CursorLocation = adUseServer
rs.CursorType = adOpenForwardOnly
rs.LockType = adLockReadOnly

db.Errors.Clear
rs.Open
If db.Errors.Count > 0 Then
   WriteError

Else
   StatusBar1.SimpleText = "Response: " & _
      FormatNumber(rs.Fields(0).Value, 0)

End If

End Sub
```

Thoughts on Opening a Recordset Object

Of the three approaches I talked about in the last two chapters to create a recordset, the two I described in this chapter are the ones I use, typically. I like the simplicity of just using the `Recordset` object and the `Connection` object to access the database. I also like the ability to open the recordset with the options Especially, I really want the options for the type of cursor and locking strategy. Of course, I can't use this technique to execute any commands that have parameters, which is why I use the other approach. As with the other approach, I have the freedom to set all of the key values in the `Recordset` object before I open it.

Summary

In this chapter you learned the following:

- You can use a server-side cursor when you want to manipulate the recordset on the database server.
- You can use a client-side cursor when you want to manipulate the recordset on the database client.
- You can use read-only locks to prevent others from changing the data while your recordset is open.
- You can use pessimistic locks to prevent others from changing the values in the current record in your recordset. This technique however incurs a significant amount of overhead in the database server.
- You can use optimistic locking when you don't expect others to change the data in the current row while you are editing it. Of course you have to handle the error condition that may arise if someone does change the data while you are accessing it.
- You can use batch optimistic locking when making changes in a group of records. This approach has the least overhead of all of the locking methods, but requires you to verify each row you changed to insure that the data wasn't changed by another database client before you made your changes.

✦ ✦ ✦