UP TO THIS POINT I’ve written a lot about extracting and massaging data from your data sources, but hardly a word about saving changes to the database or simply transporting the resultset data to some other tier. I expect that most of you work with applications where you need to change the data from time-to-time. This chapter focuses on the mechanics of persisting your changes to a data source or transporting your in-memory DataTable or DataRow data. Remember that ADO.NET is basically an “in-memory” data manager so “updating” might be as simple as changing the Value property in a DataRow; deleting a row could be implemented by using the RemoveAt method against the Rows collection. Updating might also be as complex as calling a series of conditional SQL queries that update a related set of tables applying complex business rules and triggers under a strict permissions regimen with convoluted referential integrity constraints.

In some architectures you might have another tier write to a database or simply process the information you send back; that’s why I talk about transport mechanisms in this chapter. Using this approach you have to maintain the DDL metadata to ensure that the receiving code can figure out which row to update or delete. ADO.NET can assist with this process in a number of ways—as I explain in this chapter.

Because ADO.NET does not (yet) support pessimistic locking or any type of server-side cursor or state management, the number of update strategies available to you is somewhat limited when compared with ADOc. Yes, even with this and other limitations you can still save your data, however, these restrictions mean that you have to design your systems from the outset to support locking strategies that take this lack of page- or row-locking into account. In other words, you can’t depend on strategies that lock down one or more database rows/pages/tables while you make changes to the data. Instead, you must figure out how to “request” changes and expect collisions—unless you devise a query strategy that reduces or even eliminates collisions.

If you’ve read any of my previous books, you’ll know that I’ve been encouraging these designs for some time. Yes, there are some (quite a few) situations where locking is an integral part of the design; if you’re sure your design falls into this category, ADO.NET is probably not the best solution for you. Yes, you can attempt to use ADOc from your .NET application—it still supports pessimistic
locking just as it always has—but I'm hesitant to encourage anyone to build new ADOc applications at this point in time. I just hate to see you waste your time with ADOc development when there are new versions of ADO.NET Data Providers being developed that should provide all of this missing ADOc functionality.

Changing Data—The Fundamentals

I was well into this chapter when I realized that I had left off one little, but very important, detail: how to change the data in a DataTable. That is, how to take a value from a variable (perhaps provided by the user) and save it into a DataRow Value property. I've already discussed the data structures involved—the DataTable, DataRow, and Rows collection—so you have probably already guessed how to write to the DataRow object’s Value property. Addressing the Value property is a little different when compared with ADOc—so let's revisit it.

If the DataSet is strongly typed (see Chapter 16), you can refer to the Value property with far less code (because you provided additional code earlier to expose the DataSet as an object). But let's start with an untyped DataSet.

NOTE One important difference between ADOc and ADO.NET is the concept of the "current row." When you address rows in ADO.NET you must refer to a specific DataRow object or a specific member of the Rows collection by number; there is no current row pointer as there is in ADOc. That said, I think it would be interesting to have ADO.NET support bulk change operations against an entire (filtered) set of rows—perhaps in a future version.

Addressing the Value Property with Untyped Datasets

After you have a new row, or if you're dealing with existing rows, you can simply reference the Value property using a variety of syntaxes. Note that the Value property is assumed in the following examples. I understand that default properties are legal when the property is parameterized.

myRow(0) = "Fred"
myRow(1) = "Soccer Coach"