Beginning ASP.NET 2.0 Databases

From Novice to Professional

Damien Foggon

Apress®

Beginning ASP.NET 2.0 Databases: From Novice to Professional

ISBN-13 (pbk): 978-1-59059-577-0

ISBN-10 (pbk): 1-59059-577-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Hassell
Technical Reviewers: Ronald Landers, Sahil Malik
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editors: Marilyn Smith, Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Susan Glinert
Proofreader: Linda Seifert
Indexer: Julie Grady
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 2

Introducing Relational Databases

In Chapter 1, you learned that you can use almost any kind of data source to drive your dynamic pages. You also learned that the relational database is the most commonly used type of data source for this purpose. In this chapter, you'll take a closer look at how relational databases actually work. You'll see how data is organized inside a database, and to make sure you understand it, you'll build your own sample database from scratch.

If you're wondering why this whole chapter focuses on the theory and setup of a database, it's because a badly designed and badly built database will nearly always come back to bite you once you start to build an application that uses it. And this isn't an issue specific to ASP.NET. Any application on any platform that connects to a database will run into problems if the database isn't designed correctly in the first place. Applications become harder to expand and slower to run as you spot-fix individual problems that wouldn't have come up if you had built the database correctly.

A data-driven Web site relies totally on its data source for content, so having it drag its heels because you built it wrong wouldn't be a good thing. You'll look further at the actual designing of a database in Chapter 13, but you need to be familiar with the basics before you can go there.

This chapter covers the following topics:

- The different pieces that make up a database: the tables that hold the data, the columns that define what the table holds, and the rows that contain the actual data
- The different types of SQL queries and an introduction to the most commonly used queries, as well as stored procedures
- How indexes can make your databases more efficient
- The different types of relationships between tables and how these can be modeled in the database
- How to create and use SQL Server 2005 database diagrams, which show the layout of your database
- An introduction to database views and triggers

The Databases and Tools

We're going to work with three different databases throughout the course of this book. You've already seen one of these, Microsoft Access at the end of Chapter 1. The other two that we'll look at are Microsoft SQL Server 2005 Express Edition and MySQL 5.0.

Microsoft SQL Server 2005 Express Edition is the free version of Microsoft's full SQL Server 2005 database server. The Express Edition lacks several features available in the full-blown version (none of which will cause any problems during the course of this book), and the numerous administration tools SQL Server 2005 comes with aren't included. In previous releases of SQL Server, the lack of administration tools for the free edition was a problem. However, a quite powerful administration tool, SQL Server Management Studio Express Edition, was released along with the SQL Server 2005 Express Edition. We'll use this tool to manage the SQL Server 2005 Express Edition database that we will build in this chapter.

The second, real, database that we'll use is MySQL 5.0. This is a free database server that, in its latest incarnation, provides a very comprehensive range of functionality that rivals other databases. Several free administration tools work with MySQL. We'll use of one of these: MySQL Query Browser.

As you saw in Chapter 1, Visual Web Developer 2005 Express Edition provides functionality for interacting with databases, although that functionality is limited. Visual Web Developer 2005 Express Edition allows you to connect to any data source that has an ODBC driver or OLE DB provider. If you're using ODBC or OLE DB to connect to the data source, you can only view and query the data that it contains. If you're using the SqlClient data provider—that is, connecting to SQL Server—then Visual Web Developer 2005 Express Edition offers most of the same functionality provided by SQL Server Management Studio Express Edition.

You can find the instructions for installing SQL Server 2005 Express Edition, SQL Server Management Studio Express Edition, MySQL 5.0, and MySQL Query Browser in Appendix A. You can find the complete details of the sample database you'll be building in this chapter in Appendix D. It's a simple database, which contains the details of several MP3 Players, their Manufacturers, and supported Formats.

■ **Note** As marketing people are prone to do, they make the job of talking about their products quite long-winded. What sort of name for a product is Visual Web Developer 2005 Express Edition? If I had to use that every time I talked about the product, this book would be about three times longer! I'm going to use shorter versions of the names: Visual Web Developer, SQL Server 2005, and SQL Server Management Studio. Just keep in mind that I'm referring to the Express Editions.

Tables, Rows, and Columns

The first thing to know and be comfortable with is that a relational database stores all data as *tables*. Each of these tables represents a single, distinct subject: an object or an event. For example, a table may contain details of Manufacturers (as in Figure 2-1), fish, or compact discs. Or it may keep data on appointments, deliveries, or customer service enquiries.

	ManufacturerID	ManufacturerName	ManufacturerCountry	ManufacturerEmail	ManufacturerWebsite
	1	Apple	USA	lackey@apple.com	http://www.apple.com
	2	Creative	Singapore	someguy@creative.com	http://www.creative.com
	3	iRiver	Korea	knockknock@iriver.com	http://www.river.com
	4	MSI	Taiwan	hello@miscomputer.co.uk	http://www.miscomputer.co.uk
	5	Rio	USA	Greetings@rio.com	http://www.rio.com
more rows...					

Figure 2-1. *A simple table*

In general, databases shouldn't store information about several types of objects or events—say, cats and fish—in the same table, unless the application of the database says otherwise. Biologists, for example, will want to keep details on cats and details on fish separate. More than likely, the details they keep for the two species of animal will be quite different. On the other hand, an online pet store may use a single table to keep a record of all the pets it has in stock. Cats and fish would be grouped together as "pets" in one table.

When you create a table in a database, you give it a name to reflect its contents—Book, Compact_Disc, Customer_Service_Enquiry, and so on. The table in Figure 2-1 is named Manufacturer. If you start calling a table Cats_And_Fish, for example, chances are you actually want to be creating two tables: one for cats and one for fish.

Every table contains a number of *rows*, or *records* if you prefer (or even *tuples*, if you're a mathematician). Each row represents exactly one instance of the object or event the table holds details about. In Figure 2-1, each row in the Manufacturer table holds the details for exactly one Manufacturer. These details aren't duplicated or continued elsewhere in the table, so when you locate that particular row, it contains all the information you have on that Manufacturer. In Figure 2-2, for example, the row containing all the information about Creative has been highlighted.

	ManufacturerID	ManufacturerName	ManufacturerCountry	ManufacturerEmail	ManufacturerWebsite
	1	Apple	USA	lackey@apple.com	http://www.apple.com
▶	2	Creative	Singapore	someguy@creative.com	http://www.creative.com
	3	iRiver	Korea	knockknock@iriver.com	http://www.river.com
	4	MSI	Taiwan	hello@miscomputer.co.uk	http://www.miscomputer.co.uk
	5	Rio	USA	Greetings@rio.com	http://www.rio.com
more rows...					

Figure 2-2. *A row in a table contains data about one object instance.*

Every row contains a number of *columns*, also called *attributes* or *fields*. Each column contains a single piece of information indicated by the column's name. Like the name for a table, the name for a column should be as unambiguous as possible. In the example in Figure 2-2, all of the columns are prefixed with *Manufacturer*. If it used a column called Name to represent the Manufacturer's name, for instance, it would not be immediately clear what it's referring to—Name could refer to a person's name, a Web site's name, or any name you like if you take it out of context. With the column name ManufacturerName, it's pretty clear what this column will contain.

■ **Note** All table and column names should start with a letter and be followed only by more letters, numbers, or an underscore—never a space. Some, but not all, databases permit using a few punctuation characters in names, but it's easier to stay clear of them altogether.

Retrieving information from a table is reasonably simple, because every table must contain a column or a combination of columns that uniquely identifies any piece of data in the table. This means that it doesn't matter in what order you add rows to the table, because you'll still be able to identify them individually. When you're building a database table, you identify this column or combination of columns as the table's *primary key*. In Figure 2-1, for example, the ManufacturerID column does this job nicely. Because of this primary key, you can access any column in a database with relative ease, as long as you know the column name, the value of the primary key for the row it's in, and the name of the table. For example, say you need a contact e-mail address for iRiver. To get this information, you need to find the Manufacturer table, then the row for iRiver, and then the value in the ManufacturerEmail column in that row, as shown in Figure 2-3.

	ManufacturerID	ManufacturerName	ManufacturerCountry	ManufacturerEmail	ManufacturerWebsite
	1	Apple	USA	lackey@apple.com	http://www.apple.com
	2	Creative	Singapore	someguy@creative.com	http://www.creative.com
▶	3	iRiver	Korea	knockknock@iriver.com	http://www.river.com
	4	MSI	Taiwan	hello@miscomputer.co.uk	http://www.miscomputer.co.uk
	5	Rio	USA	Greetings@rio.com	http://www.rio.com
more rows...					

Figure 2-3. *Pinpointing data in a database*

Every table should have a primary key. It doesn't have to be an ID number (although that's the norm in a simple table) as in this example, but you must be able to guarantee that each value for that primary key column will be unique. A person's last name or an appointment date won't do for a primary key, but a global unique identifier (GUID) or a product's Amazon standard identification number (ASIN) should do fine. Consider the situation where a table doesn't have a primary key. The database server may not be able to identify a specific row in a table, so it might return the wrong one or return many. What if a Web site were trying to retrieve a user's preferences and presented him with the wrong set of options? What if credit card numbers weren't unique but were used as primary keys? You could get sent the wrong bill or be charged with someone else's transactions. You can see why primary keys must be unique.

You can also use a combination of columns, rather than just one column, as a primary key. If a primary key is a single column, it's a *simple primary key*. If it consists of two or more columns, it's a *composite primary key*. For example, you couldn't uniquely identify an album in a table by its name alone (consider 4—the name of albums by Peter Gabriel, Led Zeppelin, and Black Sabbath, no less), so you could set the table's primary key to contain both the band and title. You'll see further examples of composite primary keys in the "Many-to-Many Relationships" section later in this chapter.

You can create a new table in a database in many ways, depending on which database server software and which development environment you're using. In the following sections, you'll investigate how to do this within SQL Server 2005 and MySQL 5.0.

Try It Out: Creating a Table in SQL Server 2005

In this example, you'll create the Manufacturer table shown in Figure 2-1 inside a new SQL Server 2005 database using the tools provided in SQL Server Management Studio. Follow these steps:

1. Start SQL Server Management Studio. You're immediately presented with the Connect to Server dialog box.

2. Enter the server name as `localhost\BAND` and select SQL Server Authentication as the authentication method. Enter a Login of **sa** and a Password of **bandpass**. Check the Remember Password check box. Your dialog box should look like Figure 2-4. Once all the information is entered, click the Connect button to connect to the database server.

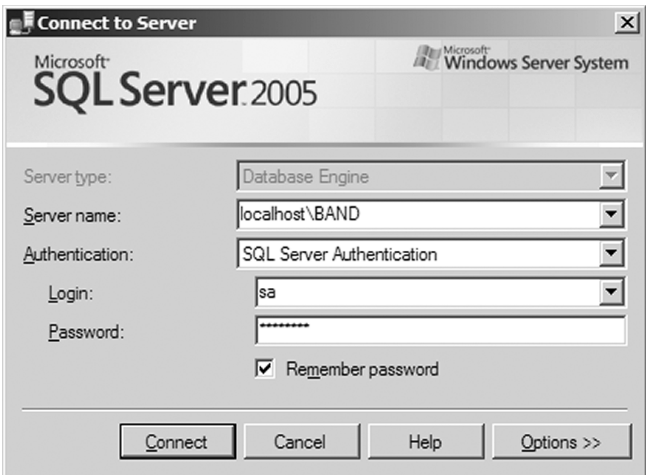

Figure 2-4. *Connecting to the correct SQL Server instance*

3. In the Object Explorer, right-click the Databases node and select New Database from the context menu to open the New Database dialog box.

4. Enter a name of **Players** for the database, as shown in Figure 2-5, and click the OK button to create the database. This will close the dialog box, and you'll see in the Summary window that the database has been created.

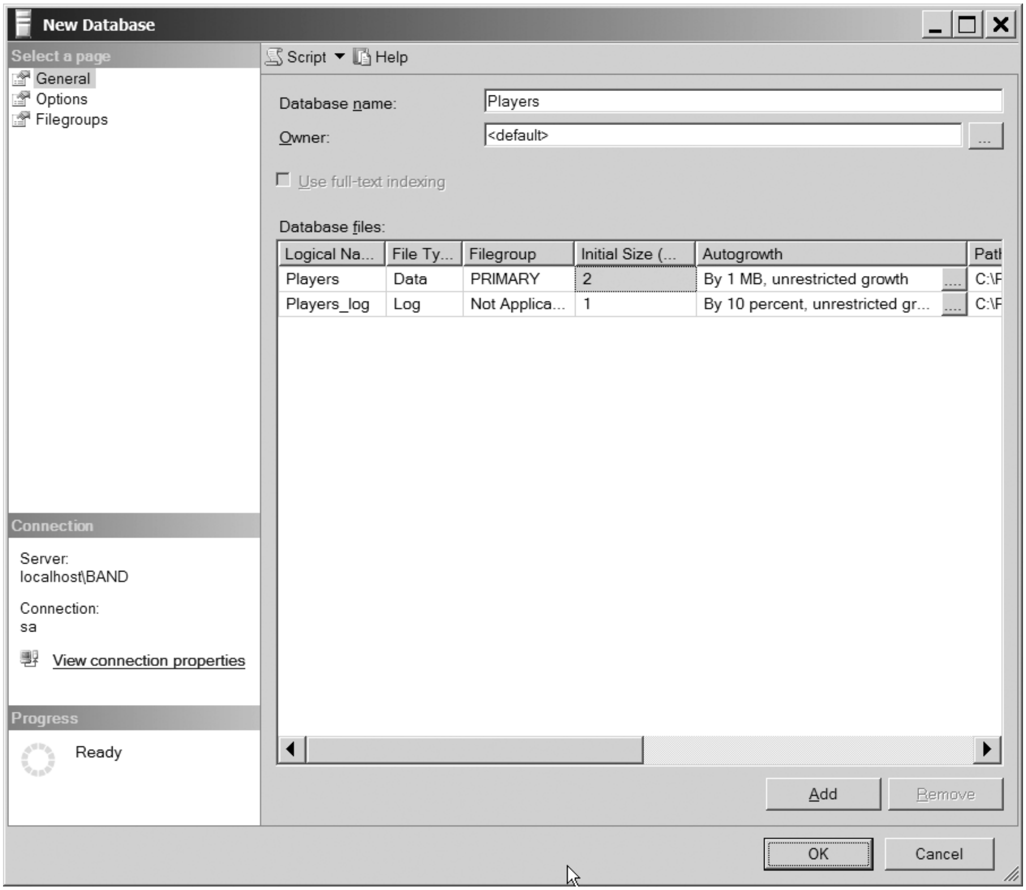

Figure 2-5. *The New Database dialog box*

5. You can also see that the database has been created by expanding the Databases node in the Object Explorer, as shown in Figure 2-6. Expand the new database, right-click the Tables node, and select New Table.

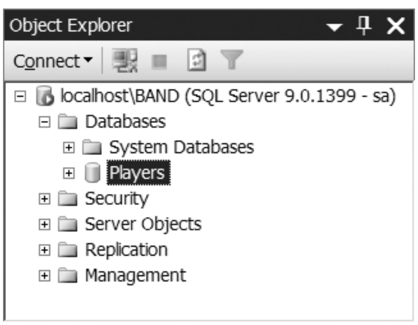

Figure 2-6. *The Players database has been created.*

6. The first task is to create the ManufacturerID column that contains the primary key for this table. Enter **ManufacturerID** in the Column Name field and select int from the drop-down list for the Data Type. Finally, uncheck the Allow Nulls check box. Your column definition should look like Figure 2-7.

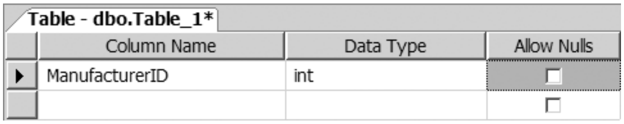

Figure 2-7. *The basic column details have been entered.*

7. Right-click the column you just created and select Set Primary Key. This marks the column as the primary key for the table. This is indicated by the key icon in the column to the left of the Column Name field, as shown in Figure 2-8.

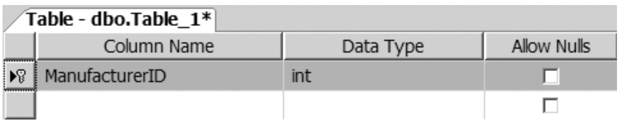

Figure 2-8. *Primary key columns are indicated graphically.*

8. Now add the four remaining columns to the Manufacturer table, as shown in Figure 2-9. As you enter details for a column in the table, a new blank row is added to the bottom of the list, allowing you to enter the details for the next column. Set up the remaining four columns as follows:

Column Name	Data Type	Allow Nulls
`ManufacturerName`	Varchar(50)	Not checked
`ManufacturerCountry`	Varchar(50)	Checked
`ManufacturerEmail`	Varchar(100)	Checked
`ManufacturerWebsite`	Varchar(100)	Checked

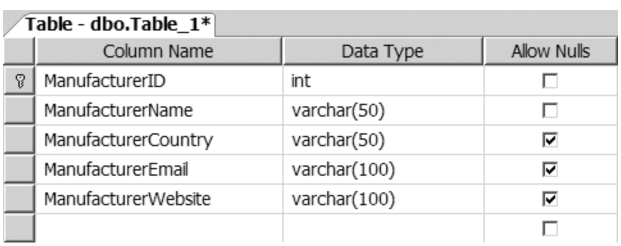

Figure 2-9. *The column definitions for the Manufacturer table in SQL Server 2005*

9. In the Properties window for the table (shown on the right side of the SQL Server Management Studio window), select ManufacturerID as the Identity Column.

10. Save the table by clicking the Save button in the toolbar or by selecting File ➤ Save Table_1 from the menu. In the Choose Name dialog box, enter the name **Manufacturer**, as shown in Figure 2-10, and then click the OK button.

Figure 2-10. *Entering a name for the new table*

11. To confirm that the table has been created, expand the Tables node in the Object Explorer. You'll see the new Manufacturer table in the list of tables (you can expand this to show the full details of the table).

12. Open Visual Web Developer and switch to the Database Explorer view. Right-click the Data Connections node and select Add Connection from the context menu. In the Add Connection dialog box, enter the same information as you used to connect in SQL Server Express Management Studio in step 1. In addition, select Players from the Connect to a Database drop-down list. Once you've entered the information, click OK to add the connection.

13. Expand the connection you just added, and then expand the Tables node in the tree that is presented. You'll see the Manufacturer table that you've just created, as shown in Figure 2-11.

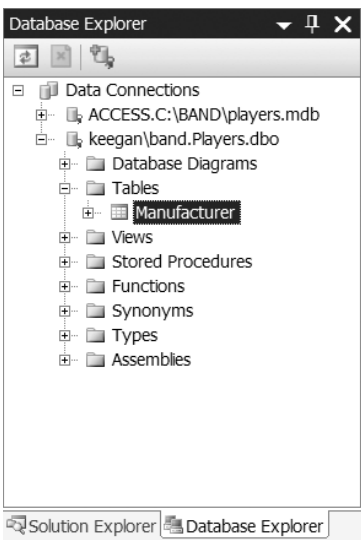

Figure 2-11. *Visual Web Developer can also connect to the SQL Server 2005 database.*

How It Works

As you've just seen, the tools provided by SQL Server Management Studio make creating a new table within a database quite simple. In the example, you actually accomplished three different tasks:

- Creating a new database
- Creating a new table in the database
- Creating new columns in the table

When first launching SQL Server Management Studio, you first connect to a SQL Server 2005 database server. Once you're connected to the database server, you can create a new database quite easily. You specify the name of the database, and SQL Server Management Studio takes care of the details of creating the database and adding it to the list of databases that you can see.

Once you've created the database, you can add new tables to it as required. You can see how easy it is to create a column by simply specifying the column's name and data type, and deciding whether the column can have null values.

You also saw how you can create a primary key on a table very easily by using the context menu for the column. As you learned earlier, primary keys allow you to individually identify a row in a table. You'll see how important primary keys are when we look at relationships between database tables later in the chapter.

The final piece in the puzzle was the setting of the identity column for the table. This allows you to set an auto-incrementing column in the table, which will be incremented by one every time a new row is added to the table. If you look again at the drop-down list that is presented for the table (you can view and edit the design of an existing table by selecting Modify from the table's context menu in the Object Explorer), you'll see that the only column that is available as an identity column is ManufacturerID. ManufacturerID can be an identity column because it contains only integers, and these can be incremented automatically every time a new row is added. As you also have a requirement for ManufacturerID to be your primary key—with unique values—setting it as the identity column makes perfect sense.

SQL Server Management Studio actually interprets your wishes to create databases and tables into queries in the database's own language—Structured Query Language (SQL)—which you'll look at in the "Queries and Stored Procedures" section later in this chapter. Actually, you have several other ways to send these same queries and build the Manufacturer table, including the following:

- Using Visual Web Developer to interact with the SQL Server 2005
- Using the SQL Server 2005 client tools to work with the database
- Using a database's command-line utility, such as `osql.exe`, as you'll see in Chapter 11
- Building your own application to send the query to the database

At the end of the example, you saw how to connect Visual Web Developer to the database you created. From within the development environment, you can see the structure of the databases, view the contents of tables, and run queries against the tables. Since you connected to the database using the SqlClient data provider, the context menu also provides options to modify the database. In later examples, you'll use Visual Web Developer to look at the database, but you will not use its management features. You'll stick to using SQL Server Management Studio for making modifications.

Try It Out: Creating a Table in MySQL 5.0

In this example, you'll create the Manufacturer table shown in Figure 2-1 inside a new MySQL 5.0 database using the tools provided in MySQL Query Browser. Follow these steps:

1. Start MySQL Query Browser. You're immediately presented with the Connect to MySQL Server Instance dialog box.

2. Enter a Server Host of **localhost**, a Username of **root**, and a Password of **bandpass**. Your dialog box should look like Figure 2-12. Once you've entered the information, click OK to connect to the database server.

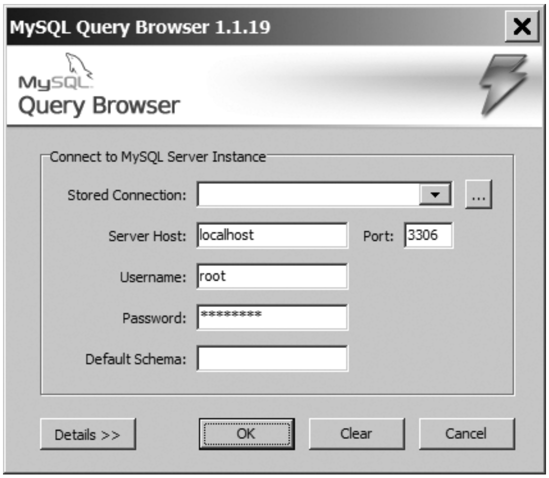

Figure 2-12. *Connecting to the correct MySQL instance*

3. Right-click in the Schemata pane on the right side of the MySQL Query Browser window and select Create New Schema from the context menu to open the Create New Schema dialog box.

4. Enter a name of **Players** for the database, as shown in Figure 2-13, and then click the OK button. This will close the dialog box, and you'll see that the new database has been added to the Schemata pane, as shown in Figure 2-14.

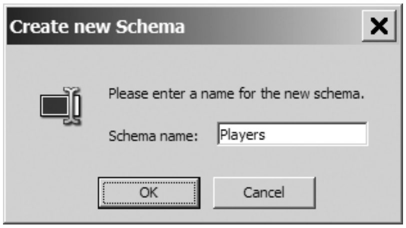

Figure 2-13. *The Create New Schema dialog box*

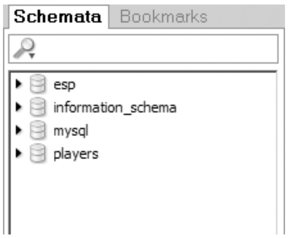

Figure 2-14. *The Players database has been created.*

5. Right-click the Players database and select Create New Table from the context menu.

6. The first step is to create the ManufacturerID column that contains the primary key for the table. Double-click in the first row of the Columns and Indices grid underneath Column Name and enter **ManufacturerID** as the name. You'll see that the column has already been marked as the primary key and is set to be auto-incrementing.

7. Accept the defaults for the rest of the values. Your column definition should look like Figure 2-15.

Figure 2-15. *The ManufacturerID column has been created.*

8. Now add the remaining four columns to the Manufacturer table, as shown in Figure 2-16. As you enter details for a column in the table, a new blank row is added to the bottom of the list, allowing you to enter the details for the next column. Set up the four columns as follows:

Column Name	Datatype	Not Null
ManufacturerName	Varchar(50)	Checked
ManufacturerCountry	Varchar(50)	Not checked
ManufacturerEmail	Varchar(100)	Not checked
ManufacturerWebsite	Varchar(100)	Not checked

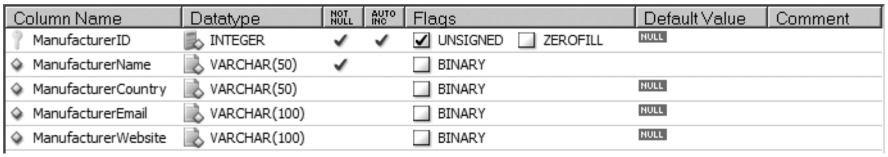

Figure 2-16. *The column definitions for the Manufacturer table in MySQL 5.0*

9. Enter a table name of **Manufacturer** at the top of the Table Editor dialog box, and then click Apply Changes to save the table. In the Confirm Table Edit dialog box, click Execute to create the table. Once the table has been saved, click Close to close the Table Editor dialog box.

10. To confirm that the table has been created, expand the Players database in the Schemata pane. You'll see the new Manufacturer table in the list of tables (you can expand this to show the columns that make up the table).

11. Open Visual Web Developer and switch to the Database Explorer view. Right-click the Data Connections node and select Add Connection from the context menu.

12. Click the Change button next to the Data Source field, and then select Microsoft ODBC Data Source from the Change Data Source dialog box.

13. Select the Use Connection String option and enter the following connection string:

```
Driver={MySQL ODBC 3.51 Driver};server=localhost;database=players;
```

14. Enter a Username of **root** and a Password of **bandpass**.

15. Click OK to add the connection.

16. Expand the connection that you just added, and then expand the Tables node in the tree that is presented. You'll see the Manufacturer table that you just created, as shown in Figure 2-17.

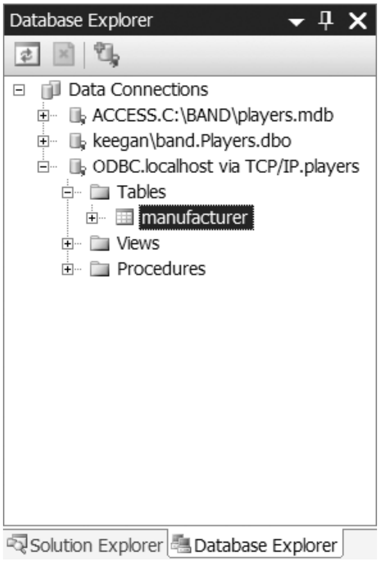

Figure 2-17. *Visual Web Developer can also connect to the MySQL 5.0 database.*

How It Works

As you saw, the tools for MySQL Query Browser are comparable to those available using SQL Server Management Studio, although they work a bit differently.

As with SQL Server Management Studio, the creation of the database and the table are performed in SQL. When you created the table, you had a sneak preview of this SQL in the Confirm Table Edit dialog box. You'll look at the SQL to modify the database structure in Chapter 11.

At the end of the example, you saw how you can use Visual Web Developer to view a MySQL database. Using the SqlClient data provider, the process is quite simple, and Visual Web Developer takes care of a lot of the work for you. However, when you use the ODBC data provider, you must create the connection string yourself. Here, you specified the following connection string:

```
Driver={MySQL ODBC 3.51 Driver};server=localhost;database=players;
```

You need to tell the ODBC data provider which ODBC driver you want to use, and then specify the specific properties for the driver—in this case, the server and database to which you're connecting.

The same is also true when you need to use the OLE DB data provider. You specify the OLE DB provider to use, and then set any properties that are specific for that provider. If you wanted to connect to the Access database, you would use the following connection string:

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\BAND\Players.mdb
```

Once the connection has been made, Visual Web Developer treats all non-SqlClient data sources equally. You can view and query the database, but you cannot modify the database in any way. The options to modify the database are simply not displayed in any of the context menus.

■ **Note** Each type of ODBC driver and OLE DB provider has a different syntax for its connection string. A good place to look for the connection string that you're after is `http://www.connectionstrings.com`. It has connection string examples for the common ODBC drivers and OLE DB providers.

Column Properties and Constraints

When building even a simple table, you can do a lot more than give the columns in the table a name. You can give each one a series of properties that strongly types and then further restricts the range of values that it can hold. This is akin to the way you give every variable in C# a simple type, or perhaps even a complex type if you want to restrict its values further.

Strictly speaking, you actually give each column a set of *properties* and then apply zero or more *constraints* that restrict the values it can hold.

The following are the column properties and constraints you've seen in the Manufacturer table example:

Column Name: This is the name of the column.

Data Type: This is the data type of the column. In the Manufacturer table, you used only two types—int/integer and varchar—but there are many more. You can find a complete list of data types you can give to a column in Appendix B.

Length: When you specified a data type of varchar, you also specified a number, in brackets, that indicates the maximum number of characters that may be entered in the column. The length property is available only for data types that contain text, such as char and varchar.

Allow Nulls: If you allow null values for a column, you're saying that the column can actually be completely empty, with nothing in it. And by "nothing," I do mean nothing. Spaces, zeros, or any other actual characters are not the same as having a null value in the column.

Primary Key: This sets whether the column is part of the primary key for its table. For the Manufacturer table, you have one column making up the primary key, indicated by the key icon next to the column name. As mentioned earlier, it is possible to have multiple columns making up the primary key, and all the columns in the key would be indicated in the same way.

You also set another property on the ManufacturerID column, making it an identity column and giving it an auto-incrementing value. This is where you'll see the first difference between SQL Server 2005 and MySQL 5.0.

In SQL Server 2005, you specify that the ManufacturerID column is auto-incrementing by setting the Identity Column property for the table. SQL Server Management Studio shields you from some of the details of designing tables, and this is one of those cases. When you set the table's identity column, you're actually modifying the ManufacturerID column directly and setting three properties on that column:

Is Identity: This indicates that the column is an identity column and will have an auto-incrementing value.

Identity Seed: This sets the value given to the first row entered into the table. The default is 1.

Identity Increment: This sets the number added to the most recently created row in the table to produce the next value of the column for a new row yet to come. The default is 1.

By default, an automatically generated integer column will be set to 1 for the first row created in the table, then 2 for the second, 3 for the third, and so on. If you set Identity Seed to 10 and Identity Increment to 2, the first row would get 10, the second 12, the third 14, and so on.

You can actually see these values for the column if you look at the Column Properties tab at the bottom of the main SQL Server Management Studio window and expand the Identity Specification node, as shown in Figure 2-18.

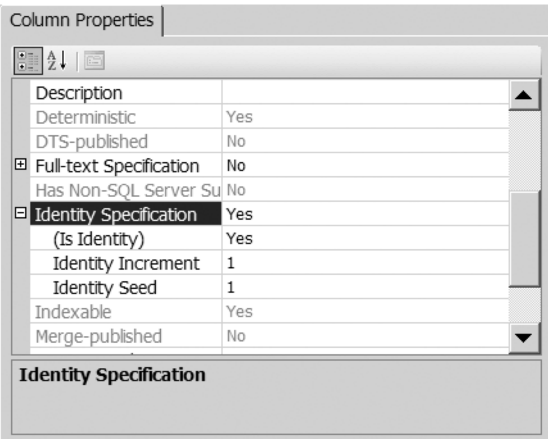

Figure 2-18. *Viewing the Identity Specification details for a column*

MySQL 5.0 specifies that a column is auto-incrementing by setting the AUTO INC property when adding the column. If you look back at Figure 2-16, you'll see that the AUTO INC column is checked for ManufacturerID.

■ **Note** You can have only one auto-incrementing column on a table. Although SQL Server Management Studio and MySQL 5.0 specify auto-incrementing columns differently, they both allow you to have only one column set as auto-incrementing at a time. SQL Server Management Studio allows you to select one column from the Identity Column list, and MySQL Query Browser lets you check only one AUTO INC column at a time.

Once you've set the general characteristics for a column, you can narrow them down by applying constraints. We've already looked at two constrains: Allow Null and Primary Key. The Column Properties tab allows you to set a further constraint on the column: its default value.

The default value for a column is specified in the Default Value or Binding property for SQL Server 2005 and as the Default Value for MySQL 5.0. This property allows you to specify a value that is used when the user doesn't enter a value when creating a new row.

You'll learn about one more type of constraint in this chapter. The foreign key constraint is concerned with maintaining the validity of data between two tables that are related. Of course, this doesn't make much sense until you understand how tables can be related, so you'll come back to this topic in the "Relationships Between Tables" section.

Queries and Stored Procedures

Many parallels exist between the way you program in .NET and the way you set up a relational database. You've already come across the idea of strongly typing columns and giving them properties. It shouldn't be a surprise then to learn that relational databases also allow you to perform actions (methods, if you like) on rows, columns, tables, and even the database itself. These actions are written in SQL, and knowing how to write queries in SQL is as fundamental to working with a database as knowing how tables, rows, and columns fit together.

Recall from Chapter 1 that the second step of talking to a database is to send it a query to retrieve, create, modify, or delete some data. This query needs to be defined and called explicitly, much like a method in .NET, and you use SQL to do this.

So, what is SQL, and why do you need to use it? Why can't you just use C# instead? SQL (usually pronounced "sequel") is the de facto standard language for talking to relational databases. Invented by IBM along with the original idea of relational databases, SQL was designed to fit the mathematical concepts that relational databases were built upon, while being straightforward to use. (Interestingly, the man who first published the rules for relational databases, Dr. E. F. Codd, disliked SQL quite a lot and preferred another query language.)

SQL is now in its second version as an International Organization for Standardization (ISO) and American National Standards Institute (ANSI) standard. It essentially works like ODBC as a common interface to a database that all vendors implement and everyone uses. One key difference between SQL and ODBC is that although all vendors implement the majority of the basic elements of the SQL standard, they then add their own proprietary queries to it and badge the whole as their version of SQL. Microsoft SQL Server uses Transact-SQL (T-SQL), Oracle uses PL\SQL, and MySQL aims to implement straight ANSI-standard SQL, although it hasn't managed all of it just yet.

SQL Queries

SQL queries are at the heart of what you'll be doing in the rest of this book. They're the commands the page gives the database, and they can be sent individually or in batches. You need to learn how to write those queries correctly, what kind of results they will return, and how to handle those results.

SQL can be divided into three main parts: a Data Manipulation Language (DML), a Data Definition Language (DDL), and a Data Control Language (DCL). Over the next few pages, you'll take a whirlwind tour through the key queries that you'll use throughout this book. I'll expand on the syntax for each instruction as you use them, and Appendix C provides a summary syntax reference for these queries.

■ **Note** Keywords in SQL aren't case-sensitive. However, I write them in all uppercase letters to make them easily distinguishable from the values you add to queries.

DML Queries

SQL's DML contains queries that let you create, retrieve, update, and delete data from a database. It has the following four basic queries:

`INSERT`: The `INSERT` query creates a new row in a table, and then adds some new data to it. For example, here is the query to add a new row to the Manufacturer table:

```
INSERT INTO Manufacturer (ManufacturerName, ManufacturerCountry,
  ManufacturerEmail, ManufacturerWebsite)
VALUES ('Apple', 'USA', 'lackey@apple.com'', 'http://www.apple.com')
```

Each column in the row you want to give a value to is named in the first list, and the values they will be given are in the second list, respectively. If the table has an identity column (as the Manufacturer table does), you don't need to specify it, as it is added automatically with the new row.

`UPDATE`: The `UPDATE` query changes the values of one or more columns in a table row. For example, to change the name of a Manufacturer from Apple to Pear, issue the following query:

```
UPDATE Manufacturer
SET ManufacturerName = 'Pear'
WHERE ManufacturerName = 'Apple'
```

As mentioned earlier, every column can be identified uniquely using the table name, primary key value, and column name. `UPDATE` queries can use all three to pinpoint exactly which piece of data to change, but they can also effect more sweeping changes, modifying several rows at a time by being less specific.

`DELETE`: The `DELETE` query removes a row or rows from a table. For example, to remove the entry in the Manufacturer table for the Manufacturer Pear, issue the following query:

```
DELETE FROM Manufacturer
WHERE ManufacturerName = 'Pear'
```

Like `UPDATE`, `DELETE` can target many rows in a table at a time. You need to be careful using `DELETE`. One false step, and you might delete all the data in a table by accident.

`SELECT`: The `SELECT` query fetches data from the database and returns it to the waiting page. For example, to retrieve a list of all Manufacturers and their e-mail addresses, issue the following query:

```
SELECT ManufacturerName, ManufacturerEmail
FROM Manufacturer
```

The `SELECT` query is incredibly powerful. You can use it to preprocess data, retrieve data across several tables at once, and then work on that data again before the page gets it. You can return tables of data or single values. You can present data using aliases or using a column's name as it is in the table. You can filter the results that you're returning using the `WHERE` clause.

Note There are books dedicated to just the `SELECT` query, so don't be disheartened if you don't get your `SELECT` queries working first time round. One of the best books about `SELECT` queries is *SQL Queries for Mere Mortals* by Michael J. Hernandez and John L. Viescas (0201433362: Addison-Wesley, 2000).

The basic syntax for all four queries is pretty straightforward. To begin with, you'll just plug in values to those simple queries and go. Then you'll start to vary and tweak. You can already see that the `WHERE` keyword is used in `UPDATE`, `DELETE`, and `SELECT` queries. It can match more than one row if you want to affect more than one row and can concatenate conditions together with Boolean operators (`AND`, `NOT`, and `OR`) to create specific clauses that may not match any rows at all.

DDL Queries

A DDL query lets you build, alter, and remove databases, tables, relationships, constraints, indexes, and more. For example, the sample database that you'll build in this chapter can also be built using a mixture of DDL and DML: DDL to create the Players database, construct the tables within it, and the columns within the tables, and some DML to add the values to the tables.

DDL has three basic queries:

`CREATE`: The `CREATE` query allows you to create a new database or object within the database. For example, to add a new table called Player with PlayerID and PlayerName columns, issue the following query:

```
CREATE TABLE Player (PlayerID INT, PlayerName VARCHAR(50))
```

The `CREATE TABLE` query is quite powerful. You can create as many strongly typed columns for the table as you like, specify a primary key, and set some of the column properties and constraints you saw earlier.

`ALTER`: The `ALTER` query allows you to modify a database object that already exists. For example, to add a new column called PlayerStorage to the Player table, issue the following query:

```
ALTER TABLE Player ADD PlayerStorage VARCHAR(50)
```

It's possible for a database to refuse to execute an `ALTER` query and return an error. This is usually because in changing the table, constraint, and so on, the altered version of the database will break the rules that still apply to the database and violate its integrity. Or rather, it will render the data invalid. For example, changing the type of the ManufacturerName column in the Manufacturer table to integer isn't allowed.

`DROP`: The `DROP` query allows you to delete any object in a database. For example, to delete the Player table from the database, issue the following query:

```
DROP TABLE Player
```

As with `ALTER`, a database may not execute a `DROP` query if the altered version of the database breaks its integrity rules.

■ **Caution** As long as your page has the appropriate privileges to delete a database, the server will go ahead and delete anything you tell it to if it doesn't violate a constraint, regardless of whether you've backed up anything or the database still contains data. Database servers have no concept of a recycle bin either, so once you say delete, it's gone. Be very careful using `DROP`. It can kill anything—database, table, constraint, and so on—just as `CREATE` and `ALTER` can create and modify anything.

DCL Queries

All database servers can also restrict which of the previous SQL queries a user may execute. The DCL queries are used to control access to the database. The following are the three most common DCL queries:

`GRANT`: The `GRANT` query allows you to give a user account the permission to run a certain kind of SQL query. For example, to let the user account Damien `INSERT` and `SELECT` data from the Manufacturer table, issue the following query:

```
GRANT INSERT, SELECT ON Manufacturer TO Damien
```

`DENY`: The `DENY` query allows you to prevent a user account from running a certain SQL query that it already has permission to run indirectly, say, because the permission was given to a group or role to which the user was assigned. For example, to prevent the user account Jill from running `DELETE` and `DROP` queries against the Manufacturer table, issue the following query:

```
DENY DELETE, DROP ON Manufacturer TO Jill
```

`REVOKE`: The `REVOKE` query completely removes the permission to run a certain SQL query from a user account. For example, to remove all permissions from an exEmployee user, issue the following query:

```
REVOKE ALL FROM exEmployee
```

ALTERNATIVES TO SQL

SQL won't be going anywhere for quite some time to come, because it's too well established and because it's the de facto standard language that all database servers use. Indeed, millions of lines of SQL run every day. But that doesn't mean you'll always need to use it.

With the release of SQL Server 2005, Microsoft has moved the goal posts quite a bit by allowing queries to be written in any of the .NET languages (C#, VB.NET, and so on), as well as in traditional SQL. For more information about using a .NET language this way, visit the SQL Server 2005 Web site at `http://www.microsoft.com/sql/2005/default.mspx`. You can also refer to *Pro SQL Server 2005* by Louis Davidson (1-59059-477-0; Apress, 2005).

If you think XML may be your calling, then you also have a third option. The World Wide Web Commission (W3C) has been working on an XML-based database querying language for some time. XQuery is still a working draft—the biggest the commission has ever created—but will be pretty solid when it's finished. The big companies such as Microsoft, IBM, and Oracle are all working on this with the W3C, so it will be well supported. For more details, go to `http://www.w3.org/XML/Query`.

Stored Procedures

Most relational database servers allow you to store SQL queries along with the databases they query. These are known as *stored procedures*, and they allow you to insulate the application developer from the intricacies of your database. After all, if all the developer is after is a list of Manufacturers from the database, does she really need to know that the table is called Manufacturer and then issue the `SELECT` query against that table to return the required columns?

Stored procedures allow you to create something like a `GetManufacturers` stored procedure and let the developer use that. Rather than the `SELECT` query itself, the page now sends a call to a stored procedure on the database, along with any parameter values it may require, just as you call a method on an object.

You'll look at using stored procedures in much greater detail in Chapter 10.

Indexes

While constraints help ensure that any modifications to the database don't disturb the validity of the data it contains, and so potentially slow down the rate at which you interact with a database, the aim of indexes is to increase the rate at which you can retrieve information. Consider a situation where you want to find all the references to SQL Server in this book. You could read this book from cover to cover and write them all down, or (if the publisher has done a particularly good job) you could turn to the back of the book, look in the index under SQL Server, and turn to the pages listed under that entry. The second method—using an index—is obviously a lot faster, and a database index works in the same way for the same reason.

Consider a situation where you want to retrieve information about all the Manufacturers who are based in Japan. Even with just ten Manufacturers, the database must work through all of the rows to make sure it has found all the entries that fit the criteria before returning the results to the page. As you can see in Figure 2-19, this search returns only two Manufacturers.

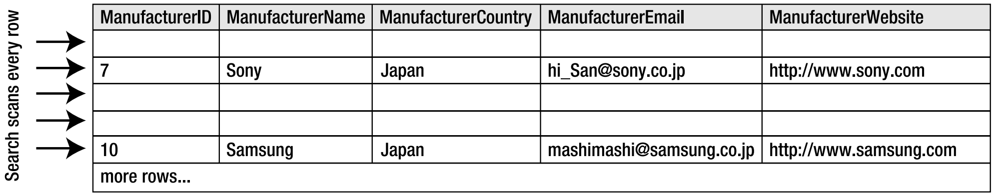

Figure 2-19. *Scanning through a table without an index*

By asking it to create an index on the ManufacturerCountry column of the Manufacturer table (the ManufacturerCountry column is to referred to as the *index key* in this context), the database server makes available to any searches an ordered list of the values in the ManufacturerCountry column. Essentially, this works in the same way the index in the back of a book works. When it needs to look up a Manufacturer, the search knows that titles are ordered alphabetically in the index you've created, so it just looks under *J*, finds the Manufacturers that have Japan as the location, and follows the index links to the correct rows. Rather than search through all of the rows in the Manufacturer table, it looks through two, as shown in Figure 2-20.

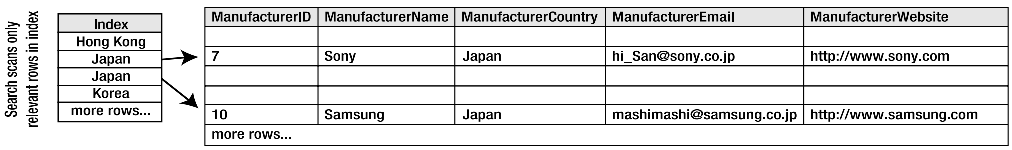

Figure 2-20. *Scanning through a table with an index*

Database indexes work exclusively behind the scenes, and aside from adding and removing indexes, you never need to reference them in your code. If an index exists it will be used automatically.

Adding the right indexes to the right tables can significantly improve performance. If you frequently issue a query that requests information to be ordered on or grouped by a certain column, it makes sense to add an index to the database based on that column.

Of course, there are always downsides. The database server must maintain every index added to the database, which means a performance hit if items are frequently added, deleted, or changed. With each modification, the server must first make that change, see if it affects any index, and then update the index if it does. That's three operations per modification. An index also consumes a fair amount of additional disk space. Therefore, overusing indexes has downsides, especially when they contain large amounts of data.

The power of indexes is in creating them wisely. For example, the effectiveness of an index whose index key column contains values that are usually the same will be much less than one where the values are unique. Consider also that a database server silently copies all the values in a nonclustered index (the default type, as described in the next section) key column in order to sort them and maintain the index. Therefore, choosing a column for the index key that contains sizable values (in other words, values that require a lot of storage) will increase the resources needed by the database and make it slower to use. The same is true of indexing a column that changes regularly, as every change requires the index to be altered.

Where possible, you should choose integer columns as indexes over those that are text-based. You should also avoid adding indexes on columns that change regularly.

Types of Index

You can add several kinds of indexes to a database:

Simple index: A simple index uses only one column as the index key.

Composite index: A composite index uses two or more columns in its index key.

Nonclustered index: A nonclustered index contains a list of index key columns in the correct order with links to the actual rows in the table (see Figure 2-20).

Clustered index: This is the most important kind of index in a database. It determines the order in which rows in a table are stored in the database. Because the clustered index changes the ordering of the rows in the table, you can have only one clustered index per table. Creating a primary key column in a table automatically creates a simple clustered index using the primary key column as the index key.

Unique index: A unique index ensures that values in the index key columns are unique, as well as orders them.

Simple and composite indexes are mutually exclusive, but you can create nonclustered, clustered, and unique indexes with one or more columns in the index key.

In the sample database, you'll add a simple index to the Manufacturer table using the ManufacturerCountry column as the index key. With only a few records in the table itself, this will have a small effect on performance, but it's important to know how to add indexes to your databases.

Try It Out: Adding Indexes in SQL Server 2005

In this example, you'll add a simple index to the ManufacturerCountry column in the Manufacturer table of the database using SQL Server Management Studio. Follow these steps:

1. Start SQL Server Management Studio. Connect to the `localhost\BAND` server using the login details that you used in the first example.
2. Expand the Databases node in Object Explorer. Expand the Players database, and then expand the Tables node.
3. Right-click the Manufacturer table and select Modify from the context menu.
4. Right-click in the table definition window and select Indexes/Keys from the context menu to open the Indexes/Keys dialog box.
5. Click Add to create a new index. Under the Identity grouping, enter **IX_ManufacturerCountry** as the index's Name, as shown in Figure 2-21.

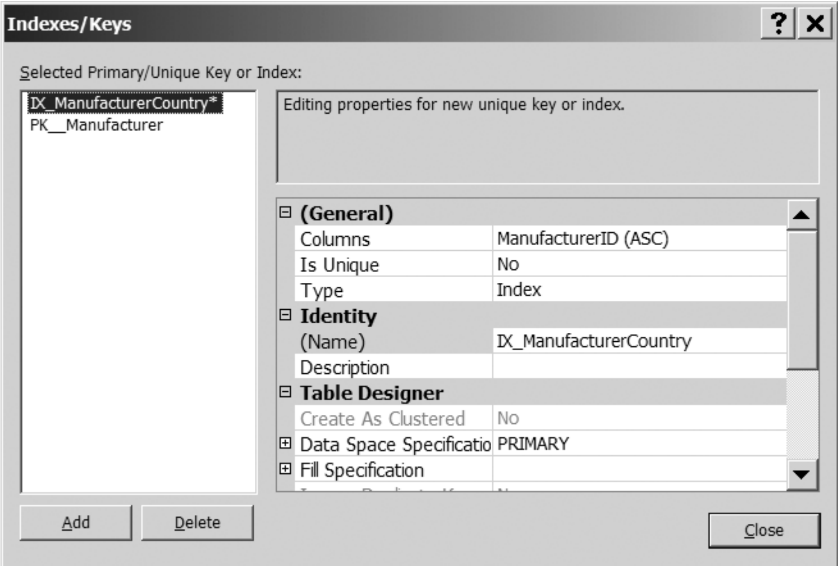

Figure 2-21. *The Indexes/Keys dialog box*

6. Click the Columns property under the (General) grouping, and then click the ellipsis in the right column.
7. In the Index Columns dialog box, select ManufacturerCountry from the Column Name drop-down list, as shown in Figure 2-22.

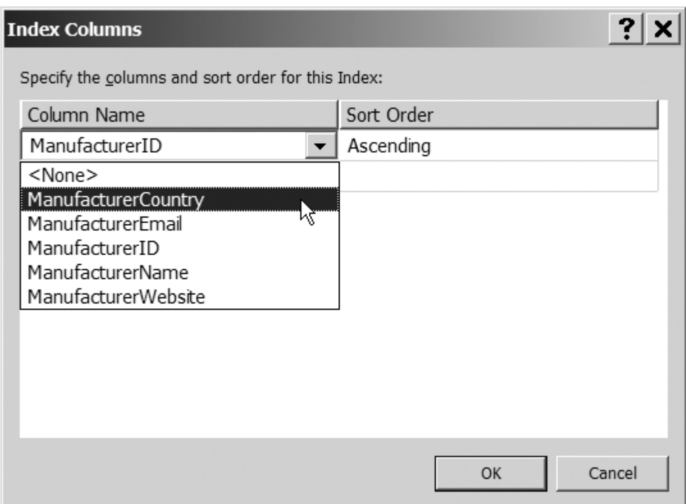

Figure 2-22. *Selecting the column for an index*

8. Click OK to close the Index Columns dialog box.
9. Click Close to close the Indexes/Keys dialog box.

10. Click Save on the toolbar to save the changes to the database.

11. In the Object Explorer, expand the Manufacturer table node, and then expand the Indexes node, as shown in Figure 2-23. (You may need to refresh the node, by right-clicking Indexes and select Refresh from the context menu, to see the two indexes that are present on the table.)

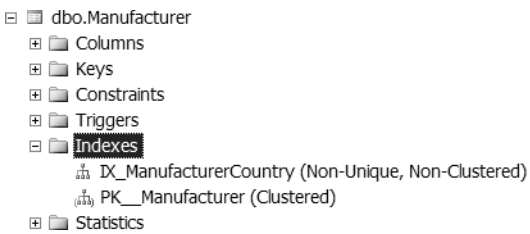

Figure 2-23. *Indexes can be viewed in Object Explorer.*

How It Works

As you can see, creating an index using SQL Server Management Studio is very simple. You give it a name, and then specify the columns that you want to include in the index. SQL Server Management Studio deals with sending the correct queries to the database to create the index. We'll look at the SQL queries for creating database indexes in Chapter 11, but for the curious, what is actually sent is as follows:

```
CREATE INDEX IX_ManufacturerCountry
ON Manufacturer (ManufacturerCountry)
```

If you need to create a clustered or unique index, you use the query `CREATE CLUSTERED INDEX` or `CREATE UNIQUE INDEX`, respectively. A nonclustered index is the default for the `CREATE INDEX` query, but you could have used `CREATE NONCLUSTERED INDEX` to accomplish the same thing.

In the same way, the indexes you create from the Indexes/Keys dialog box are nonclustered by default. The Indexes/Keys dialog box allows you to create all three index types (nonclustered, clustered, and unique), and it provides a bit of logic to prevent you from making any simple errors. In Figure 2-21, you see two options that allow you to alter the index that you're adding: Is Unique and Create As Clustered. However, since you already have a clustered index on the table (for the ManufacturerID primary key), the Create As Clustered option is disabled.

You can also delete indexes by selecting the index in the Indexes/Keys dialog box and selecting Delete.

■ **Note** You'll notice that in the very last step of this example, you explicitly clicked the Save button to save the changes to the database. SQL Server Management Studio allows you to modify the database tables and doesn't automatically save changes to the database structure. This prevents any errors that you may make from affecting data that you don't really want to change. To keep the changes that you've made, you must explicitly save the changes. If you try to close the table without doing so, SQL Server Management Studio will prompt for you to save the changes.
