1

Database Fundamentals

Before you start to look at accessing databases from C# code, there are a few basics that you need to know. It is necessary to have a formal definition of what is meant by the term database, and that's the first thing you examine in this chapter. Once you have this definition, you look in more depth at the features that databases (and, more specifically, database management systems) offer, and see the difference between relational and object-oriented database management systems. Next, you investigate many of the commonly used database management systems. Finally, you are introduced to the language used to manipulate databases, Structured Query Language (SQL). Along the way you learn the terminology used by databases, see how databases may be represented graphically, and get your first look at the database management system used in this book — SQL Server 2005 Express Edition.

If you've had any previous experience with databases, you may find that you are already familiar with much of the material in this chapter. However, this information has been included so you can avoid any ambiguities and common misconceptions that might cause problems later. Whatever your level of experience, it is well worth recapping the basics to ensure a strong foundation of knowledge for later chapters, and this chapter will also serve as a reference for you later on. Remember, get a firm grasp of the basics and the rest will come easily.

In this chapter, you learn:

- ❑ What databases are
- ❑ The terminology used for databases
- ❑ The features are offered by database management systems
- ❑ What database management systems are available
- ❑ How to manipulate data in a database

Chapter 1

What Is a Database?

It's fair to say that most computing applications make use of data in one form or another, whether in an obvious way or with more subtlety. In most cases this data is *persistent*, which means that it is stored externally to the application and doesn't disappear when the application isn't running. For example, the following applications obviously store and manipulate data, in small or large quantities:

- An application used by a shop to keep records of products, sales, and stock
- An application used to access human resources information in a large enterprise
- A web page that allows the retrieval of historical currency conversion rates

It is a little less obvious whether the following applications use stored data, but it is likely that they do:

- Any web page you care to examine — many store some if not all the information they display in external locations.
- Web pages that don't display stored data, but that track user activity.
- Online games with persistent worlds where player and character information is stored in a centralized online location.

Of course, it's also difficult to say exactly how these applications store their data. It is possible that the software developers store data in text files on a hard disk somewhere, in computer RAM, or even on stone tablets that are manually transcribed to a computer terminal when requested by a user. It is far more likely, however, that they store data in a database.

The *Oxford English Dictionary* defines the word "database" as follows:

> *1. A structured collection of data held in computer storage; esp. one that incorporates software to make it accessible in a variety of ways; transf., any large collection of information.*

This definition alone goes some way to describing why a database is better than, for example, storing text files. A key word here is *structured*. Simply storing large amounts of data in an unstructured way, such as text files, is usually referred to as *flat-file storage*. This has many disadvantages, including the problem of compatibility when proprietary storage formats are used, the inability to locate specific information stored somewhere in the middle of the data, and the general lack of speed in reading, editing, and retrieving information.

Databases provide a standardized way to store information, and do so in a manner that promotes extremely fast access by hundreds, even thousands, of users, often simultaneously.

A phone book, for example, contains the names of individuals (or organizations), phone numbers, and possibly addresses. Flat-file storage might involve no ordering whatsoever — perhaps a bucket containing each name/phone number/address combination on a separate piece of paper (which would make retrieval of any specific one an interesting challenge at best). More likely, there would be some organization, typically by the first letter of people's last names, which is a step up from a bucket of data, but still lacking finesse. This data might make up the basis of a flat-file storage format (each record in order, encoded in some machine-readable or human-readable way), but can more accurately be called a *directory*. Directories are data stores that are organized in a way to optimize data retrieval in one specific mode of use. In this example it is easy to find the phone number of someone as long as you have his name, but the inverse scenario does not apply. If you have a phone number and want to know to whom it belongs, you won't find a phone book particularly useful. While it is technically possible, it's not something you'd want to do unless you have a lot of time on your hands. And it would be a lot easier to simply dial the number and ask to whom you were speaking. Even in flat-file storage, searching for a specific entry still means starting at the beginning and working your way through to the end in some systematic (and arbitrary) way.

Databases store data in a highly structured way, enabling multiple modes of retrieval and editing. With phone book data in a database, any of a number of tasks would be possible in a relatively simple way, including the following:

- Retrieve a list of phone numbers for people whose first name starts with the letters "Jo."
- Find all the people whose phone numbers contain numbers whose sum is less than 40.
- Find all the people whose address contains the phrase "Primrose" and who are listed with full names rather than initials.

Some of these operations might require a little more effort to set up than others, but they can all be done. In most cases they can be achieved by querying the database in certain ways, which means asking for the data in the right way, rather than manipulating data after it has been obtained using, say, C#.

Structure and efficiency aren't the only benefits offered by a database. With a centralized, persistent data store you have many more options, including simple data backup, mirroring data in multiple locations, and exposing data to remote applications via the Internet. You look at these and other possibilities later in this chapter.

Before I continue, one thing must be made abundantly clear from the outset. The word "database" does not — repeat, not — refer to the *application* that stores data. SQL Server, for example, is not a database. SQL Server is a database management system (DBMS). A DBMS is responsible for storing databases, and also contains all the tools necessary to access those databases. A good DBMS shields all of the technical details from you so that it doesn't matter at all where the data is actually stored. Instead, you just need to interact with whatever interfaces the DBMS supplies to manipulate the data. This might be through DBMS-supplied management tools, or programmatically using an application program interface (API) with C# or another programming language.

In fact, there are different types of DBMS to consider. The two most important and well known are:

- Relational database management systems (RDBMSes)
- Object-oriented database management systems (OODBMSes)

In the next sections you explore the differences between these types and the key features of both.

Relational Database Management Systems

Relational database management systems (RDBMSes) are what you would typically think of as a DBMS, and these are the most commonly found and often used systems. SQL Server, for example, is an RDBMS. Two things are essential for an RDBMS:

- Separate tables containing data
- Relationships between tables

The following sections examine tables; relationships; an important property of relational databases that emerges from these constraints — normalization; and one of the underlying mechanisms by which all this is achieved: keys.

Tables

Characteristically, an RDBMS splits the data stored in a database into multiple locations, each of which contains a specific set of data. These locations are called *tables*. A table contains multiple *rows*, each of which is defined in multiple *columns* (also known as *records* and *fields*).

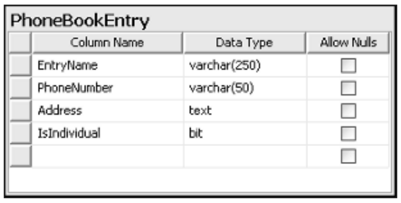
Phone book data, for example, could be stored in a single table, where each row is a single entry containing columns for name, phone number, and address. Tables are defined such that every row they contain includes exactly the same columns — you can't include additional columns for a given row just because you feel like it. Columns are also assigned specific data types to restrict the data that they can contain. In this example all the data types are strings, although more space might be allocated to address fields because addresses typically consist of more data than names. You might decide to include an additional Boolean column in a phone book table, however, which would say whether the record was an individual or an organization. Other tables might include numeric columns, columns for binary data such as images or audio data, and so on. In addition, a table can specify whether individual columns must contain data, or whether they can contain null values (that is, not contain values).

Each table in a database has a name to describe what it contains. There are many conventions used for naming tables, but the one used in this book is to use singular names, so for a phone book table you'd use `PhoneBookEntry` for the name rather than `PhoneBookEntries`.

The name and structure of tables within a database, along with the specification of other objects that databases may contain and the relationships between these objects, are known as the *schema* of the database.

The word "object" is used here with caution — but correctly. Relational databases can contain many types of objects (including tables), as you will see throughout this book, but that doesn't make them object-oriented. This distinction will be made clearer in the section on OODBMSes shortly.

Figure 1-1 shows the supposed `PhoneBookEntry` table graphically, by listing column names and data types, and whether null values are permitted.



PhoneBookEntry

Column Name	Data Type	Allow Nulls
EntryName	varchar(250)	☐
PhoneNumber	varchar(50)	☐
Address	text	☐
IsIndividual	bit	☐
		☐

Figure 1-1: The PhoneBookEntry table

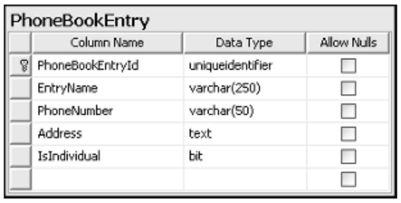
The diagram for the `PhoneBookEntry` table shows data types as used in SQL Server, where some data types also include lengths. Here, the `EntryName` field is a string of up to 250 characters, `PhoneNumber` is a string of up to 50 characters, `Address` is a string with no defined maximum size, and `IsIndividual` is a `bit` (0 or 1), which is the SQL Server type used for Boolean data.

Keys

Within database tables it is often important to uniquely identify rows, especially when defining relationships. The position of a row isn't enough here, because rows may be inserted or deleted, so any row's position might change. The order of rows is also an ambiguous concept because rows may be ordered by one or more columns in a way that varies depending on how you are using the data in the table — this is not a fixed definition. Also, the data in a single column of a table may not be enough to uniquely identify a row. At first glance, you might think that the `EntryName` column in the `PhoneBookEntry` table example could uniquely identify a row, but there is no guarantee that the values in this column will be unique. I'm sure there are plenty of Karli Watsons out there, but only one of them is writing this book. Similarly, `PhoneNumber` may not be unique because people in families or student housing often share one phone. And a combination of these fields is no good, either. While it is probably quite unlikely that two people with the same name share a phone, it is certainly not unheard of.

Without being able to identify a row by either its contents or its position, you are left with only one option — to add an additional column of data. By guaranteeing that every row includes a unique value in this column, you'll always be able to find a particular row when you need to. The row that you would add here is called a *primary key*, and is often referred to as the PK or ID of the row. Again, naming conventions vary, but in this book all primary keys end with the suffix `Id`.

Graphically, the primary key of a table is shown with a key symbol. Figure 1-2 shows a modified version of `PhoneBookEntry` containing a primary key.



PhoneBookEntry

	Column Name	Data Type	Allow Nulls
🔑	PhoneBookEntryId	uniqueidentifier	☐
	EntryName	varchar(250)	☐
	PhoneNumber	varchar(50)	☐
	Address	text	☐
	IsIndividual	bit	☐
			☐

Figure 1-2: The PhoneBookEntry table with a primary key

The data type used here is `uniqueidentifier`, which is in fact a GUID (Globally Unique IDentifier). It is not mandatory to use this data type, but it is a good thing to use. There are many reasons for this, including the fact that GUIDs are guaranteed to be unique (in all normal circumstances). Other typically seen types for primary keys include integer values and strings.

It is not always absolutely necessary to define a new column for primary key data. Sometimes the table contains a column that is unique by definition — a person's Social Security number for example. In some situations, combining two columns will give a unique value, in which case it is possible to use them to define a compound primary key. One example of this would be the combination of postal code and house number in a table of U.K. addresses. However, it is good practice to add a separate primary key anyway, and in this book most tables use `uniqueidentifier` primary keys.

Relationships

RDBMSes are capable of defining relationships between tables, whereby records in one table are associated (linked) with records in other tables. When storing large quantities of data, this relational aspect is both important and extremely useful. For example, in a sales database you might want to record both the products on sale and the orders placed for products. You can envisage a single table containing all this information, but it is far easier to use multiple tables — one for products, and one for orders. Each row in the orders table would be associated with one or more rows in the products table. An RDBMS would then allow you to retrieve data in a way that takes this relationship into account — for example, using the query "Fetch all the products that are associated with this order."

Relationships between items in different tables can take the following forms:

- One-to-one relationship: One row in one table is associated with a row in a separate table, which in turn is associated with the first row. In practice, this relationship is rare because if a one-to-one relationship is identified, it usually means that the data can be combined into a single table.
- One-to-many and many-to-one relationships: One row in one table is associated with multiple rows in a separate table. For example, if a list of products were divided into categories (where each product was associated with a single category), then there would be a one-to-many relationship between categories and products. Looking from the other direction, the relationship between products and categories is *many-to-one*. In practice, one-to-many and many-to-one relationships are the same thing, depending on which end of the relationship you are looking at.
- Many-to-many relationship: Rows in one table are freely associated with rows in another table. This is the relationship you have in the products and orders example because an order can contain multiple products, and products can be part of multiple orders.

When considering relationships, the importance of keys is immediately obvious. Without being able to uniquely identify rows it would be impossible to define a meaningful relationship. This is because associating a row in one table with a row in another table might actually associate other rows with each other by implication.

One-to-many relationships are implemented by including a foreign key field in the table at the many end of the relationship. For example, to link products with categories you add a field to the product table that acts as a foreign key to link product rows with category rows. The value of a foreign key in a row in one table typically matches the value of a primary key in another table — in fact, the columns used for primary and foreign keys are often given the same name.

The implementation of many-to-many relationships typically involves using a third, linking table. In the products/orders example, a single row in the product table may be associated with multiple records in the linking table, each of which is associated with a single order. Conversely, each row in the order table may be associated with multiple rows in the linking table, each of which is associated with a single row in the product table. In this situation, the linking table must contain two foreign keys, one for each of the tables it is linking together. Unless required for other reasons, such as when the linking table contains additional columns relating to the linkage, or represents real data in its own right, there is often no need for you to include a primary key in the linking table.

Figure 1-3 shows four tables illustrating these relationships. Depending on how you look at it, this diagram shows three one-to-many relationships (`ProductCategory` to `Product`, `Product` to `OrderProduct`, and `Order` to `OrderProduct`), or one one-to-many relationship and one many-to-many relationship (`Product` to `Order`). To simplify things, the tables don't show column data types or whether columns are nullable.

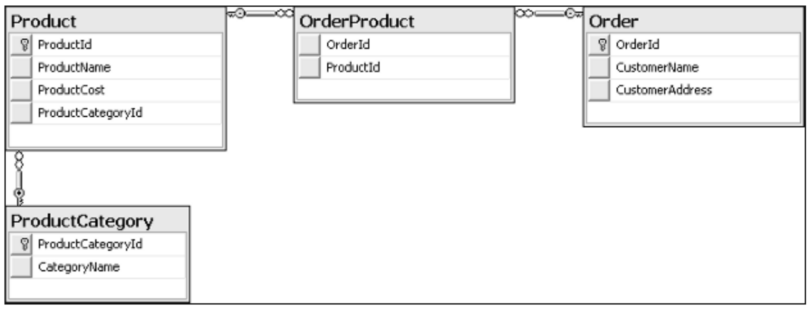


Figure 1-3: The PhoneBookEntry table with a primary key

Showing one-to-many links as a line with a key at one end and linked circles at the other (the "infinity" symbol) is just one way to display this information. You may also see lines with a 1 at one end and an ellipsis at the other. In this book, however, you'll see the format shown here throughout.

In this scheme it is typically a good idea to include a further column in the `OrderProduct` table — `Quantity`. This enables a product that appeared multiple times in a single order to be represented using a single row in `OrderProduct`, rather than several, where the number of rows would be the quantity. Without the `Quantity` column, things could quickly get out of hand for large orders!

One last thing to note here is the concept of referential integrity. Because these relationships are defined as part of the database schema, the DBMS is capable of enforcing them. This means, for example, that you can choose for the DBMS to prevent the deletion of a row that is referred to by another row. Alternatively, you could choose to have the DBMS delete any referenced rows when a row is deleted (known as a *cascaded delete*).

Normalization

Normalization is a fairly advanced topic, but one you need to be aware of from the outset. It refers to the process of ensuring that little or no data in a database is duplicated. Another way of looking at this is that it is the process of organizing the structure of data and data tables so that the most efficient method of storage is used. What happens, for example, when a single customer places more than one order? With just an order table you'd end up in a situation where customer details were duplicated because they'd need to be included in each and every order made by the customer. It would be far better to add an additional table for customers, which could be linked to multiple orders.

To extend the example: how about customers with multiple addresses? This might happen if a customer wants to send an item directly to a friend. Here, a further table containing addresses is required. But hold on — if an order is associated with a customer, and a customer has multiple addresses, how do you tell which address the order is supposed to be sent to? Clearly, even simple databases can become much more complicated quickly — and often there are multiple solutions to problems. The subject of database organization and normalization is one that you will return to many times in later chapters.

In some circumstances *redundancy*, that is, the duplication of information, can be beneficial. This is particularly true when speed is crucial because there is an overhead associated with finding a row in one table based on a foreign key in another. This may be negligible, but in large-scale, ultra-high performance applications, it can become an issue.

Object Oriented Database Management Systems

There are some situations where the integration between applications and databases must be far stronger than is possible when using RDBMSes, again mostly in high-performance applications. One approach that's been quite successful is for databases to store objects directly so that OOP applications can store and retrieve objects directly, without resorting to serialization techniques.

Because object oriented database management systems (OODBMSes) store objects directly, it is possible to manipulate data via the methods and properties of databases, and to associate objects with each other via pointers rather than the sort of relationships discussed earlier. This leads to a more navigational style of data access — getting one object can lead you to another, then another, and so on using these pointers.

Another feature of OODBMSes is that they can make use of polymorphism in much the same way as OOP programming languages — some object types inherit characteristics from a single base object type. However, other OOP features, such as encapsulation, do not mesh particularly well with the traditional view of databases, so many people dismiss OODBMSes out of hand. Nevertheless, these DBMSes have found a place in, for example, scientific areas such as high-energy physics and molecular biology.

Owing to the niche usage of these systems, and the fact that they are few and far between as well as being highly specialized, they aren't covered further in this book.

Additional Features of RDBMSes

As mentioned earlier, RDBMSes offer a lot more than the storage of data in related tables. In particular, you can rely on most to provide:

- Joins
- Functions
- Views
- Stored procedures
- Triggers
- E-mail
- Indexes
- Security
- Concurrency control
- Transactions
- Remote access
- Backups
- Mirroring and partitioning
- Management tools

In this section you look at each of these, getting a flavor for them but without going into too much depth at this stage.

Joins

In the earlier relationship discussion, it may have seemed like accessing related data from multiple tables might involve a convoluted procedure. In actual fact — luckily for us — that isn't the case. It is possible to fetch data from multiple tables simultaneously, and end up with a single set of results. The mechanism for doing this involves *joins*, of which there are several types. A join is a way to specify a relationship between two tables to obtain related data from both. A join between a product table and a category table, for example, enables you to obtain all the products belonging to a single category in one operation. This is something that you'll see in action after you've learned a bit more about the language used to execute database queries — Structured Query Language (SQL).

Functions

Any good DBMS supplies you with an extensive set of functions to use to view and manipulate data. You are likely to find mathematical functions, conversion functions, string manipulation functions, date and time manipulation functions, and so on. These enable you to perform much of your data processing inside the DBMS, reducing the amount of data that needs to be transferred to and from your applications, and improving efficiency.

DBMS functions can take several forms. There are scalar functions that return single values, table valued functions that can return multiple rows of data, and aggregate functions that work with entire data sets rather than individual values. Aggregate functions include those with capabilities to obtain the maximum value in a given column of a table, perform statistical analysis, and so on.

Another type of function that you will probably find yourself using at some point is the user-defined function. As its name suggests, you can create your own function to perform whatever task you like. User-defined functions may be scalar, table valued, or aggregate.

There is one more important feature of functions as used in SQL Server 2005 — it is possible to write them in C# code that runs (managed) inside the database. This is something you'll see in action later in this book.

Views

There are some database operations that you might want to repeat often within your applications, such as those involving joins, as detailed earlier. Rather than forcing the DBMS to combine data from multiple sources, often transforming the data along the way, it is possible to store a *view* of the data in the DBMS. A view is a stored query that obtains data from one or more tables in the database. For example, a view might be a query that obtains a list of products that include all product columns and the name of the category in an additional column. The client applications don't have to make more complicated queries involving joins to obtain this information, because it is already combined in the view. The view looks and behaves identically to a table in every way except that it doesn't actually *contain* any data; instead, it provides an indirect way to access data stored elsewhere in the database.

Apart from the obvious advantage of a view — that querying the underlying data is simplified for client applications — there is another important point to note. By telling the DBMS how the data is to be used in this way, the DBMS is capable of optimizing things further for you. It might, for example, cache view data so that retrieving its compound information becomes much faster than querying individual tables might be.

In addition, views can be defined in some quite complicated ways using functions, including user-defined functions, such that your applications can retrieve highly processed data with ease.

Stored Procedures

Stored procedures (often called *sprocs*) are an extremely important part of database programming — despite the fact that you could use a fully functioning database without ever using a sproc. Stored procedures enable you to write code that runs inside the database, capable of advanced manipulation and statistical analysis of data. Perhaps more important, their operation is optimized by the DBMS, meaning they can complete their tasks quickly. In addition, long-running stored procedures can carry on unattended inside the database while your applications are doing other things. You can even schedule them to run at regular intervals in some DBMSes.

Stored procedures don't do anything that you couldn't do by other means — for example, in C# code. However, for some operations that work with large quantities of data, it might mean transferring the data into application memory and then processing it to get a result. With stored procedures the data never has to leave the database, and only the result needs transferring to your application. With remote databases this can provide a significant performance boost.

Some DBMSes — such as SQL Server — provide you with a rich set of operations that you can use when programming stored procedures. These include cursors that you can position within sets of data to process rows sequentially, branching and looping logic, variables, and parameters. And as with functions, SQL Server lets you write stored procedures in managed C# code.

Triggers

A trigger is a specialized form of stored procedure that is executed automatically by the DBMS when certain events happen, rather than being called manually by client applications. In practice, this means defining an event that will occur at some later date ("when a new row is added to table X," for example), and then telling the DBMS to execute a certain stored procedure when that event occurs.

Triggers aren't as commonly used as some other features of DBMSes, but when they are, it's because they are the only solution to a problem, so it's good to have them. They are typically used to log or audit database access.

E-mail

Some DBMSes are capable of sending e-mails independently of other applications. This can be useful, especially when combined with triggers. It enables you to keep tabs on data in a database, as well as permitting more advanced scenarios. When orders are placed, for example, you could generate and send e-mails to customers directly from the DBMS with no external coding required. The only limitation here is that a mail server such as a simple mail transfer protocol (SMTP) server is likely to be required.

Indexes

Indexes are another way of optimizing performance by letting the DBMS know how you intend to make use of data. An index is an internally maintained table in the database that enables quick access to a row (or rows) containing specific data, such as a particular column value, a column value that contains a certain word, and so on. The exact implementation of an index is specific to the DBMS you are using so you can't make any assumptions about exactly how they are stored or how they work. However, you don't need to understand how an index is implemented to use it.

Conceptually you can think of an index as a look-up table, where you find rows in the index with a specific piece of data in one column, and the index then tells you the rows in the indexed table that match that data. To return to the phone book example, an index could be used to search for records via the phone number column instead of the name column. You would need to tell the DBMS to create an index for values in the phone number column because, by default, no indexes are created for a table other than for primary key values. By building an index based on the phone number column, the DBMS can use a much faster searching algorithm to locate rows — it no longer has to look at the phone number column of every row in the address book; instead it looks in the index (which has, effectively, already looked at every row in the address book) and finds the relevant rows.

The only downside to using indexes is that they need to be stored, so the database size increases. Indexes also need to be periodically refreshed as data in the table they are indexing changes.

The creation of indexes can be a bit of an art form. In many DBMSes it is possible to tailor indexes to closely match the queries with which they will be dealing. For example, looking for strings that end with a certain substring works well with an index built around the last 100 characters of a text column, but might not even be possible in an index built on the first 100 characters of the same column.

One commonly used type of index is the full-text index. It's useful when large quantities of text are stored in columns because the index examines the text in-depth and stores its results. This enables you to perform searches within text data much faster than would otherwise be possible because you only have to look at a word in the index rather than looking through all the text in all the columns of the original data. However, full-text indexes can require large amounts of storage.

Security

Security means a couple of things when talking about databases. For a start, it means not letting other people get access to your data. For most professional DBMSes, this isn't something you have to worry about too much. If your DBMS costs lots of money (and it probably does), you get what you pay for, and your data is secure.

The other aspect of security in databases is authorizing different users to perform different tasks. In some cases, such as in SQL Server 2005, you can approach this in a granular way. You can, for example, assign a user the rights to view data in one table but not to edit that data. You can also restrict access to individual stored procedures and control access to all manner of more esoteric functionality. Users can also be authorized to perform tasks at the DBMS level if required — such as being able to create new databases or manage existing databases.

Most DBMSes also enable you to integrate with existing forms of authentication, such as Windows account authentication. This allows for single-login applications, where users log on to a network with their usual account details, and this login is then forwarded on to the database by any applications that are used. An advantage here is that at no point does the application need to be aware of the security details entered by the user — it simply forwards them on from its context.

Alternatively, you can use DBMS-specific forms of authentication, which typically involve passing a username and password combination to the DBMS over a secure connection.

Concurrency Control

With multiple users accessing the same database at the same time, situations can arrive where the data being used by one user is out of date, or where two users attempt to edit data simultaneously. Many DBMSes include methods to deal with these circumstances, although they can be somewhat tricky to implement.

In general, there are three approaches to concurrency control that you can use, which you'll look at shortly. To understand them, you must consider an update to be an operation that involves three steps:

1. User reads the data from a row.
2. User decides what changes to make to the row data.
3. User makes changes to the row.

In all cases sequential edits are fine: that is, where one user performs steps 1–3, then another user performs steps 1–3, and so on. Problems arise when more that one user performs steps 1 and 2 based on the original state of the row, and then one user performs step 3.

The three approaches to concurrency control are as follows:

- **"Last in wins":** Rows (records) are unavailable only while changes are actually being made to them (during step 3). Attempts to read row data during that time (which is very short) are delayed until the row data is written. If two users make changes to a row, the last edit made applies, and earlier changes are overwritten. The important thing here is that both users might have read the data for the row (steps 1 and 2) before either of them makes a change, so the user making the second change is not aware that the row data has already been altered before making his change.
- **Optimistic concurrency control:** As with "last in wins," rows are unavailable only while they are being updated. However, with optimistic concurrency control, changes to row data that occur after a user reads the row (step 1) are detected. If a user attempts to update a row that has been updated since he read its data, his update will fail, and an error may occur, depending on the implementation of this scheme. If that happens, you can either discard your changes or read the new value of the row and make changes to that before committing the second change. Effectively, this could be called "first in wins."
- **Pessimistic concurrency control:** Rows are locked from the moment they are retrieved until the moment they are updated, that is, through steps 1–3. This may adversely affect performance, because while one user is editing a row, no other users can read data from it, but the protection of data is guaranteed. This scheme enforces sequential data access.

Most of the time, concurrency control is jointly handled by the DBMS and the client application. In this book, you will be using C# and ADO.NET, and data is handled in a *disconnected* way. This means that

that the DBMS is unaware of whether rows are "checked out" at any given time, which makes it impossible to implement pessimistic concurrency control. There are, however, ways in which optimistic concurrency control can be implemented, as you will see later in the book.

Transactions

It is often essential to perform multiple database operations together, in particular where it is vital that all operations succeed. In these cases, it is necessary to use a transaction, which comprises a set of operations. If any individual operation in the transaction fails, all operations in the transaction fail. In transaction terminology, the transaction is committed if, and only if, every operation succeeds. If any operations fail, the transaction is rolled back — which also means that the result of any operation that has already succeeded is rolled back.

For example, imagine you have a database with a table representing a list of bank accounts and balances. Transferring an amount from one account to another involves subtracting an amount from one account and adding it to another (with perhaps a three-day delay if you are a bank). These two operations must be part of a transaction because if one succeeds and one fails, then money is either lost or appears from nowhere. Using a transaction guarantees that the total of the money in the accounts remains unchanged.

There are four tenets of transactions that must be adhered for them to perform successfully; they can be remembered with the acronym ACID:

- **Atomicity:** This refers to the preceding description of a transaction: that either every operation in a transaction is committed, or none of them are.
- **Consistency:** The database must be in a legal state both before the transaction begins and after it completes. A legal state is one in which all the rules enforced by the database are adhered to correctly. For example, if the database is configured not to allow foreign key references to nonexistent rows, then the transaction cannot result in a situation where this would be the case.
- **Isolation:** During the processing of a transaction, no other queries can be allowed to see the transient data. Only after the transaction is committed should the changes be visible.
- **Durability:** After a transaction is committed, the database should not be allowed to revert to the state it was in before the transaction started. For example, any data added should not subsequently be removed, and any data removed should not suddenly reappear.

For the most part, you are unlikely to come across a DBMS that violates these rules, so it isn't something that you need to worry about. Transaction support in the .NET Framework is also good, and this is something you'll be looking at later in the book.

Remote Access

A good DBMS allows remote access across an intranet and, if required, the Internet. Again, this is something that most databases permit, although some configuration (of both the DBMS and firewalls) may be necessary. It is not, however, always the best option, especially when communicating with a database across the Internet.

Later in the book you'll see how an intermediary (specifically, a web service) can be used to control remote access.

Backups

After the key tasks of being able to add, edit, and delete data in a database, perhaps the most important function of a DBMS is to enable you to back that data up. However good computers are, and however good the software you use, there are some things that are impossible to predict. Hardware failure happens — it's a fact of life, albeit an unpleasant one.

You may not be able to predict a hardware or software failure, but backing up your data regularly can make things easier to cope with when the inevitable happens. Today's DBMSes put a wealth of tools at your disposal for backing up data to disk, networked storage, tape drives, and other devices. Not only that, but backups can be scheduled at regular intervals, and the retrieval of backed-up data is made as simple as it can be. This isn't a subject that is discussed in this book, because it's more of a management subject — but that's not to say that it isn't important!

Mirroring and Partitioning

Mirroring and partitioning features are similar enough to merit being covered in the same section. They both involve sharing data across multiple logical and/or physical locations.

Mirroring means configuring multiple databases to hold the same information. The databases needn't be on the same DBMS, or even on the same computer. This has important implications for backing up because it means that if one computer goes down, you have a mirrored copy of the data ready to go. Of course, this isn't a replacement for backing up — after all, it's possible that both computers could fail at the same time. However, it does provide a fail-safe option for when things go wrong and can enable you to keep client applications running while you fix the problem. Another way that mirroring databases can be useful is where extremely large numbers of clients are using the data, or where a huge amount of data is routinely transferred to and from clients. Here, mirroring databases can provide load balancing by scaling out (adding additional computers) where database queries can be distributed among computers.

Partitioning is similar to mirroring in that multiple DBMS computers may be used. Partitioning is where data that would otherwise exist in a single table is distributed among various locations. The benefits of this are not immediately obvious, but they are important nonetheless. Consider a situation in which a large enterprise with offices worldwide wants to share data across locations. Some data may well be location-specific — local customers, for example. In that case the customer data could be divided between local DBMSes in such a way that the actual implementation is totally transparent. Each location is capable of accessing all the data in the customers table as and when required, but can make speedier queries for local customer data because that is stored on its own DBMS.

Management Tools

Most DBMSes come equipped with tools to use to manipulate data and databases via a graphical user interface (GUI). However, in some cases you will have to look hard to find them. Sometimes, only third-party applications are available, and many of them aren't free. Without the luxury of a GUI administration or management tool, you have to resort to command-line tools, which often means struggling with obscure commands and tricky syntax to get even the simplest things working. Of course, expert users sometimes prefer to use command-line tools, but we mere mortals generally find it much easier to point and click to perform tasks.

A good management tool does far more than let you look at and edit data in tables. It enables you to create and manage databases, configure backups, handle mirroring and partitioning, add stored procedures and views, create database diagrams, and administer security. In this book you use the free management studio tool that is available with SQL Server 2005 Express Edition, and you'll learn how to use its features as and when you need them.

What RDBMSes Are Available?

There are dozens of RDBMSes available, each with its own horde of loyal users and each with its own take on databases. In this section, you examine a few of the more commonly used ones and learn a little about their usage.

MySQL and PostgreSQL

MySQL (`www.mysql.com`) is an extremely popular RDBMS for web site creators — in part this is because it's an open source system, and therefore effectively free to use (although you have the option of paying for a commercial license if you want). This makes it the ideal partner to use with other open source software. It's common, for example, to see web sites implemented using a combination of the Apache web server, PHP, and MySQL.

However, as with all things free there are limitations. For a start, you'll find it a real help to be familiar with an array of command-line techniques, and you can't expect installation to be as easy as running an installer application. Also, there are a lot of things that MySQL has been able to do only recently. Up until the latest release at the time of writing (version 5.0, released October 2005) views, stored procedures, and triggers weren't supported, among other things. The current release does include these, although with slightly fewer features than in other implementations. There are also lower limits on many properties of databases — such as the length of table names and such. And, more important, for large amounts of data (100GB or more) MySQL is unlikely to be happy.

MySQL is a good RDBMS to use when learning about databases, or for the small-scale hobbyist, but if security, scalability, performance, and reliability are issues, there are far better options. One other thing to be aware of is that finding a GUI to use to administer MySQL can be a complex task. There are an awful lot around, and finding one to suit you that contains all the functionality you want to use isn't easy.

PostgreSQL (`www.postgresql.org`) is another open source RDBMS, but is aimed more at the professional user. It uses a slightly different licensing model, but is still available for free.

PostgreSQL has a slightly richer feature set when compared to MySQL, although with the release of MySQL 5.0 there isn't a lot of difference. There are a few relatively minor things, such as PostgreSQL being able to partition tables. Frankly, though, if data partitioning is something you want to make use of then an open source RDBMS probably isn't the best option.

However, as with MySQL, cost-free availability is definitely an advantage, and many people find that this is all they need. It is important to be aware, however, that like MySQL, PostgreSQL won't deal with large databases well.

DB2, Oracle, and SQL Server

DB2, Oracle, and SQL Server are the three RDBMS heavy-hitters of the database world. DB2 (`www-306.ibm.com/software/data/db2`) is made by IBM — in fact, there's a whole family of DB2 products — and SQL Server (`www.microsoft.com/sql`) is made by Microsoft, which gives you as much an idea of the intended audience as anything else. Oracle (`www.oracle.com/database`) is produced by Oracle Corporation, a multinational company that's primarily known for DBMSes. All three of these RDBMSes are aimed at large-scale organizations and are optimized for large amounts of data and users.

It's actually difficult to choose among them. Apart from the platforms that they run on (SQL Server is restricted to Windows operating systems, for instance), they all contain similar capabilities. They can all, for example, contain databases whose size extends into the terabyte range. They all include sophisticated tools for backing up and for other administration. And, perhaps most important, they all include their own proprietary features — yes, some things you can do in SQL Server, you can't do in DB2, and vice versa.

Each of these DBMSes has its devotees, so the choice among them is likely to come down to whichever you were first exposed to, what you learned how to use first, or personal recommendation. (I prefer SQL Server, but that's just me.)

The important thing is that when it comes to high-end applications requiring the best security and performance, these applications can't be beat. Well, not yet in any case.

Oh, one more thing. Each of these RDBMSes costs a *lot* of money, which is why in this book you'll be using.…

SQL Server 2005 Express Edition

SQL Server 2005 Express Edition is a slimmed-down version of SQL Server 2005. It's available, for free, from Microsoft. In terms of usage, it can be difficult to tell it apart from the full version of SQL Server. This is by design. It is possible — indeed easy — to develop applications on the Express Edition and then migrate to the full version in a production environment. This is good news for a huge number of developers who might not otherwise have access to a SQL Server installation. It also makes the Express Edition a great way to learn about SQL Server.

This is not to say, of course, that there is no reason to use the full version of SQL Server 2005. The Express Edition is great for developing, and even (arguably) suitable for small web sites that don't get a lot of traffic and small applications with only a few users, but it is not nearly enough for enterprise-level applications. When it comes to an application that might have many thousands of users and terabytes of data storage, the Express Edition is simply not robust enough. For a full comparison of the features of the various editions of SQL Server 2005, see `www.microsoft.com/sql/prodinfo/features/compare-features.mspx`.

There are many other reasons for choosing SQL Server instead of another RDBMS, in particular with .NET applications. As has been noted several times, it is possible to write managed C# code that will run inside SQL Server 2005 — including SQL Server 2005 Express Edition. This is known as Common Language Runtime (CLR) integration. The .NET library also includes classes for natively accessing SQL Server data, rather than having to use a less streamlined intermediate standard such as Open Database Connectivity (ODBC). ODBC is a useful technology that provides a way to access RDBMSes with code that doesn't depend on the exact RDBMS you are using. However, in being standardized it suffers by not giving access to proprietary features, and it is slower because its instructions must be translated into native instructions.

In this book, you learn specifically about SQL Server 2005 data access with C#, using the Express Edition for simplicity (and because it's free to download and use). Please don't assume that this prevents you from learning to use other types of RDBMSes, however. In many cases, near identical code applies to other RDBMSes — and this is noted in the book where appropriate.

To work through the examples in this book, you need to install SQL Server 2005 Express Edition. You can find instructions for doing this in Appendix A, along with instructions for installing the other Express products that are used in this book should you not have them installed already.

How Do You Use a Database?

Databases are designed to be data stores that are independent of other factors, such as the programming language you use to access them. With a few notable exceptions — CLR integration, for instance — you can use them from any programming language, or even directly (via web requests, for example) in some circumstances.

Early in the history of databases, moves were made to standardize database access. In more recent times a standard has emerged (and undergone numerous revisions) — Structured Query Language (SQL). The roots of this standard can be traced back to the 1970s, with the first accepted standard (SQL-86, published in 1986) having now evolved into the most recent version, SQL:2003 (published, unsurprisingly enough, in 2003). SQL is now ubiquitous, although every major RDBMS vendor still insists on including its own proprietary "flavor" of the language. Having said that, the core commands and syntax are (more or less) the same whatever RDBMS you use. Perhaps the only remaining controversy concerns how to pronounce SQL — "see-quell" or "ess-que-ell"? Personally, I prefer the former, but feel free to differ.

At its heart, SQL is a human-readable language. In fact, without having any prior knowledge of it, you'll probably find it reasonably simple to understand in its basic form. For example, consider the following SQL command:

```
SELECT PhoneNumber FROM PhoneBookEntry WHERE EntryName = 'Geoffrey Chaucer'
```

So, what do the words that make up this command suggest? Well, it doesn't take a huge amount of working out to guess that this command will retrieve the contents of a `PhoneNumber` column from a database called `PhoneBookEntry` for a row with an `EntryName` column that has a value of `Geoffrey Chaucer`. Admittedly, `SELECT` doesn't appear to be the most obvious name of a command that would do this (`GET` or `RETRIEVE` might seem more appropriate), but don't worry — you'll get used to it. Later in this section, you encounter a SQL primer that covers the basics of what the SQL language can achieve.

Luckily for those of us who don't want to type in reams of SQL code to do anything with databases, most RDBMSes come equipped with graphical tools for administration purposes. This isn't the case with SQL Server 2005 Express Edition, although you can inspect and administer it through the Visual C# 2005 Express

Edition interface. In addition, there is a free tool — Microsoft SQL Server Management Studio Express — that mimics the more advanced administration tool that ships with the full version of SQL Server 2005. The installation of this tool is shown in Appendix A. In this book you'll use both of these techniques, and in most cases you'll be writing no more SQL code than is necessary.

It is perfectly possible to write SQL statements manually in your code, although it is often the case that you don't have to. Many .NET controls are capable of generating SQL commands automatically, which certainly saves wear and tear on your fingers. Of course, this may not give you exactly the behavior you require, and more advanced tasks will require custom SQL code, so it's still worth learning how SQL works. This book covers these controls, but you will also learn when and where you will need to write SQL code yourself.

One subject that has become increasingly important in recent years is XML, which stands for eXtensible Markup Language. This is a platform-independent standard for representing data in text files, and has become extremely popular. It does have its disadvantages — it is a flat-file data format, after all — but these are outweighed by what it makes possible. If you have had any experience using .NET, you've no doubt encountered XML. The most recent DBMSes (including SQL Server 2005 Express Edition) include additional capabilities to deal with XML data, both as a means of storage and as a format with which to retrieve data — even if that data isn't encoded as XML to start with. It is important that you cover the basics of XML here because you are likely to need to know about it before too long.

The remainder of this section covers two topics:

- ❑ A SQL Primer: The basics of the SQL language and what you can do with it.
- ❑ XML: A brief overview of the XML language and further information on its relevance in the context of databases.

A SQL Primer

This section is by no means a comprehensive guide to using SQL. In fact, entire books are devoted to the subject, including one co-written by yours truly: *The Programmer's Guide to SQL* (APress, 2003. ISBN: 1590592182). But it does teach you the basics.

There is a lot of code in this section, and you may find it useful to run these commands against a database yourself to see what is happening. However, at this stage I don't recommend doing so. You'll get plenty of experience doing this later in the book, and for now it is best just to get a basic understanding of the SQL language and see what is possible. Later you may want to refer back to this section, so don't worry if you don't take it all in the first time around.

Basic SQL Terminology and Syntax

Before getting into the SQL language itself, there are a few bits of basic terminology and syntax of which you should be aware. First, a chunk of SQL code is known (interchangeably) as a *statement*, *command*, or *query*. SQL statements might span several lines, and whitespace (spaces, tabs, new line characters, and so forth) is ignored. SQL statements consist of keywords (such as `SELECT`) and operators (`+`, `-`, and so on) combined with literal values (string constants, numbers, and so on), table and column identifiers, and often functions. SQL keywords are case-independent, but are typically written in uppercase to distinguish them from other parts of the statement. Common statements, such as those that retrieve data, are often referred to by the first keyword in the statement, so you might hear people refer to a select statement, for example. SQL statements are often made up of several parts, which are known as clauses. SQL statements may also contain embedded (nested) statements known as subqueries.

SQL statements are executed, and may return one or more results. Multiple statements can be executed as a batch, which simply means "execute statements sequentially in the order that they are written." In some SQL dialects, a semicolon (`;`) is used to signify the end of a statement, although that is generally not required, because the context is enough to tell where one statement ends and the next begins.

Retrieving Data from Tables

Data retrieval is, as you've already seen, the job of the `SELECT` keyword. There are a huge number of ways to write select statements, involving a multitude of additional keywords and techniques, and the basic result is the same in all cases — you obtain a single value (a scalar result) or zero or more rows of data. The data that's returned may not be in the same format as the data in the database, because columns may be retrieved from multiple tables, combined in some way, renamed, or processed by functions before reaching you.

Here's the simplest form that a select statement can take:

```
SELECT [Column(s)] FROM [Table]
```

In it, `[Column(s)]` is a comma-separated list of column names and `[Table]` is the table containing the columns. For example:

```
SELECT EntryName, PhoneNumber FROM PhoneBookEntry
```

This retrieves a set of rows consisting of two columns, `EntryName` and `PhoneNumber`, from a table called `PhoneBookEntry`. This is shown graphically in Figure 1-4.

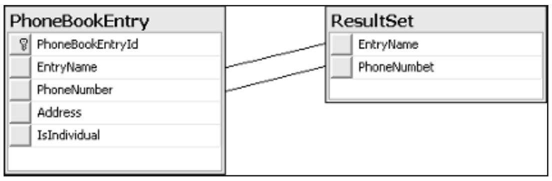


Figure 1-4: PhoneBookEntry query result set

In some circumstances, column or table names may match SQL keywords — `Name`*, for instance, is a SQL keyword and might be used as a column name. To solve this ambiguity, SQL Server enables you to use square brackets to signify that the enclosed text should not be interpreted as a SQL keyword. You could use the text* `[Name]` *in a query to refer to a column called* `Name`*, for example. Nevertheless, it is good practice not to use SQL keywords to name columns or tables so that this is not an issue.*

Often you will want to retrieve all of the data in all of the rows in a table; in that case, you can use the shorthand `*` to refer to all columns. For example:

```
SELECT * FROM PhoneBookEntry
```

It is worth noting that this shorthand notation can result in some (minor) overhead and reduction in performance because the RDBMS must work out what the columns are for you. That isn't something to worry too much about, and certainly not when prototyping code, but it may be something you return to when optimizing the performance of an application if you used it a lot when developing your code. The best practice is to avoid using `*`.

The next important thing you should know about is the capability to filter the data that you retrieve. Again, this is something you saw in an earlier example, and involves the use of the `WHERE` keyword to add a so-called where clause to the `SELECT` statement:

```
SELECT [Column(s)] FROM [Table] WHERE [Filter]
```

The filter used in a where clause may be a simple one, such as a equality between column values and a literal value:

```
SELECT EntryName, PhoneNumber FROM PhoneBookEntry
   WHERE PhoneBookEntryId = 'f4367a70-9780-11da-a72b-0800200c9a66'
```

Here data is returned for any rows that have a `PhoneBookEntry` column containing the value `f4367a70-9780-11da-a72b-0800200c9a66`. This is a GUID value, and is a unique, primary key value if this table is defined as per the example given earlier in this chapter, so the example query will return a single row of data containing the columns `EntryName` and `PhoneNumber`.

Filters can also be made up of multiple parts, combined using Boolean operators such as `AND` and `OR`, and use other comparison operators such as > for "greater than" or <= for "less than or equal to."

Another common operator in where clauses is `LIKE`, which you use to perform searches within text column values. When using `LIKE` you must supply a string literal value for comparison, which may contain wildcard symbols to widen the search criteria. In SQL Server, those symbols include `%` to refer to any string of zero or more characters, _ to refer to any single character, and others. For example:

```
SELECT * FROM PhoneBookEntry
   WHERE EntryName LIKE '%chaucer%' AND Address LIKE '%canterbury%'
```

This statement returns all the rows in the `PhoneBookEntry` table whose `EntryName` column contains the string `chaucer` and whose `Address` column contains the string `canterbury`. The query performs a case-insensitive search in SQL Server, unless the database is configured to enforce case-sensitive searching.

Combining multiple filter parts in a where clause is a flexible technique that you can use to obtain data sets that are as specific as you like. Good use of filtering can be important in the performance of your applications because minimizing the amount of data that is exchanged between your applications and your databases will lessen the time taken to make these exchanges.

Using Joins to Retrieve Data

The `SELECT` keyword can also be used to obtain data from multiple tables. There are alternative ways of doing this, of course, such as using complicated nested queries, but the simplest and most common way is to perform a join between pairs of tables in the query. There are several join types, each of which is useful in different circumstances. Briefly, the types of join you can perform are as follows (don't worry too much about these definitions at this stage — they will be explained in more detail shortly):

- ❑ Cross join: Each row in table A is joined to each row in table B.
- ❑ Inner join: Rows in table A and table B are joined according to the data contained in rows and the criteria you choose to compare these rows by. Rows may be excluded from the result set if no join is possible.
- ❑ Outer join: Similar to an inner join, but rows that would otherwise be excluded may be included in the result set depending on the exact specification of the join.

Understanding Cross Joins

The easiest of these joins to explain is the cross join, although you will probably find it the least useful. The syntax of using a cross join between, say, tables called `TableA` and `TableB`, is to specify the from clause of a query as follows:

```
FROM TableA CROSS JOIN TableB
```

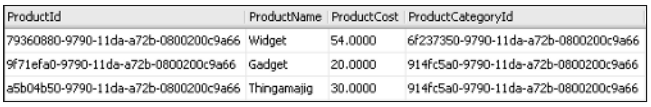
Or using the simpler syntax:

```
FROM TableA, TableB
```

The full select statement includes other clauses much like queries over a single table, including column specifications and filters. For example:

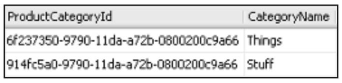
```
SELECT * FROM Product, ProductCategory WHERE CategoryName = 'Things'
```

Using actual data will help you understand the query better. Let's say that the `Product` and `ProductCategory` tables contain data as shown in Figures 1-5 and 1-6.



ProductId	ProductName	ProductCost	ProductCategoryId
79360880-9790-11da-a72b-0800200c9a66	Widget	54.0000	6f237350-9790-11da-a72b-0800200c9a66
9f71efa0-9790-11da-a72b-0800200c9a66	Gadget	20.0000	914fc5a0-9790-11da-a72b-0800200c9a66
a5b04b50-9790-11da-a72b-0800200c9a66	Thingamajig	30.0000	914fc5a0-9790-11da-a72b-0800200c9a66

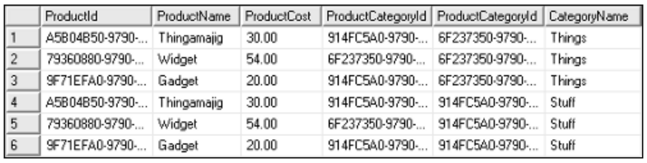
Figure 1-5: Product table contents



ProductCategoryId	CategoryName
6f237350-9790-11da-a72b-0800200c9a66	Things
914fc5a0-9790-11da-a72b-0800200c9a66	Stuff

Figure 1-6: ProductCategory table contents

Without considering the where clause for now, a cross join between these two tables results in the six rows shown in Figure 1-7. (Some of the GUID columns are truncated in this figure to save space.)



	ProductId	ProductName	ProductCost	ProductCategoryId	ProductCategoryId	CategoryName
1	A5B04B50-9790-...	Thingamajig	30.00	914FC5A0-9790-...	6F237350-9790-...	Things
2	79360880-9790-...	Widget	54.00	6F237350-9790-...	6F237350-9790-...	Things
3	9F71EFA0-9790-...	Gadget	20.00	914FC5A0-9790-...	6F237350-9790-...	Things
4	A5B04B50-9790-...	Thingamajig	30.00	914FC5A0-9790-...	914FC5A0-9790-...	Stuff
5	79360880-9790-...	Widget	54.00	6F237350-9790-...	914FC5A0-9790-...	Stuff
6	9F71EFA0-9790-...	Gadget	20.00	914FC5A0-9790-...	914FC5A0-9790-...	Stuff

Figure 1-7: Result of a cross join between Product and ProductCategory tables

There are six results because every row in `Product` (3) is combined with every row in `ProductCategory` (2): 3×2=6. Note that there are two `ProductCategoryId` columns — one from each table. This raises an important point: How do you differentiate between columns in different tables that have the same name? You do so by using an expanded reference to the columns; namely `[TableName].[ColumnName]`. The two columns here are therefore `Product.ProductCategoryId` and `ProductCategory.ProductCategoryId`. This is the method used by SQL Server and many other DBMSes to identify columns, and it's the syntax you would use if, say, you wanted to specify one of these columns in the column specification part of a select statement.

The addition of the where clause in the sample cross join query shown previously reduces the six-row result set to three rows — those with a `CategoryName` column containing the string `Things` (rows 1 to 3 in Figure 1-7).

At first glance this may seem useful, but the query hasn't really achieved much. Crucially, it hasn't taken into account the actual relationship between the rows of data. Each `ProductCategory` row is associated with a `Product` row in a one-to-many relationship, but that isn't reflected in the result set you get with a cross join. This relationship is achieved by the `Product.ProductCategoryId` column being a foreign key relating to the primary key `ProductCategory.ProductCategoryId`.

To obtain a more meaningful set of results, you need to use one of the other types of join. The simplest of them is the inner join.

Understanding Inner Joins

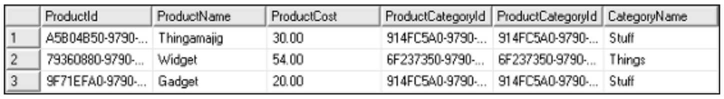
Inner joins require more information, namely a join specification, which takes a form similar to the where clause filters you saw earlier. The syntax here (omitting the column specification for now) is as follows:

```
FROM TableA INNER JOIN TableB ON [Join Specification]
```

Handily, the keywords for performing an inner join are `INNER JOIN`. To get the behavior discussed earlier, where the `Product` and `ProductCategory` tables are joined based on their respective `ProductCategoryId` columns, the query would need to be as follows:

```
SELECT * FROM Product INNER JOIN ProductCategory
   ON Product.ProductCategoryId = ProductCategory.ProductCategoryId
```

The result of this query (again with truncated GUIDs) is shown in Figure 1-8.



	ProductId	ProductName	ProductCost	ProductCategoryId	ProductCategoryId	CategoryName
1	A5B04B50-9790-...	Thingamajig	30.00	914FC5A0-9790-...	914FC5A0-9790-...	Stuff
2	79360880-9790-...	Widget	54.00	6F237350-9790-...	6F237350-9790-...	Things
3	9F71EFA0-9790-...	Gadget	20.00	914FC5A0-9790-...	914FC5A0-9790-...	Stuff

Figure 1-8: Result of an inner join between Product and ProductCategory based on ProductCategoryId columns

Adding a where clause identical to the one used earlier for the cross-join query would result in a single row being returned — row 2 in Figure 1-8.

Inner joins are powerful — even more so when you consider that they can link any number of tables together. For example:

```
SELECT [Order].OrderId, [Order].CustomerName, [Order].CustomerAddress,
      Product.ProductId, Product.ProductName, Product.ProductCost,
      ProductCategory.ProductCategoryId, ProductCategory.CategoryName,
      OrderProduct.Quantity
   FROM [Order] INNER JOIN OrderProduct
   ON [Order].OrderId = OrderProduct.OrderId INNER JOIN Product
   ON OrderProduct.ProductId = Product.ProductId INNER JOIN ProductCategory
   ON Product.ProductCategoryId = ProductCategory.ProductCategoryId
```

This may look complicated, but it really isn't. It uses lots of simple steps that look quite complex when combined. Look at the steps individually and you'll see that it's quite straightforward. Figure 1-9 illustrates the query graphically (the diamonds on the relationship lines signify inner joins — each type of join uses a different symbol).

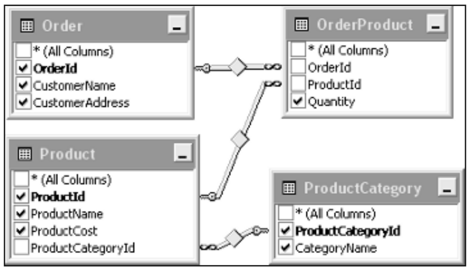


Figure 1-9: A query involving multiple inner joins

The first part of the query shows the columns to return — which are indicated by ticked boxes in Figure 1-9. `Order` is a SQL keyword, which is why it is enclosed in square brackets in the query.

The query contains three consecutive inner joins, each of which joins a pair of tables:

1. `Order` is joined to `OrderProduct` on the value of the `OrderId` columns. This results in as many rows in the result set as there are rows in the `OrderProduct` table, with row data from each row in the `Order` table being duplicated in multiple rows of the result set.
2. `ProductId` columns are used to add data from the `Product` table to the result set, and an inner join is made between `OrderProduct` and `Product`.
3. `Product` is joined to `ProductCategory` as in the previous example.

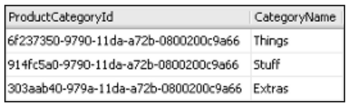
This results in a useful combination of data. The outcome gives you full order information — including what products make up an order, how many of each product are in the order, and what categories those products belong to.

You come across inner joins frequently in this book, so there are plenty more examples to get your teeth into.

You'll return to inner joins in subsequent chapters. One last note for now: It is possible to use operators other than = to join tables. For numeric fields you can, for example, use >. The results of this are more difficult to illustrate (and to explain the usefulness of), but it's something to bear in mind for later.

Understanding Outer Joins

The other types of join that you can perform are outer joins. As mentioned earlier, these are like inner joins, but are where rows that aren't joined to other rows may be included in the result set. You can see this in action if, for example, the `ProductCategory` table contains a third row, as shown in Figure 1-10.



ProductCategoryId	CategoryName
6f237350-9790-11da-a72b-0800200c9a66	Things
914fc5a0-9790-11da-a72b-0800200c9a66	Stuff
303aab40-979a-11da-a72b-0800200c9a66	Extras

Figure 1-10: ProductCategory table contents

Executing the inner join query shown earlier:

```
SELECT * FROM Product INNER JOIN ProductCategory
   ON Product.ProductCategoryId = ProductCategory.ProductCategoryId
```

gives the same results shown in Figure 1-8. The additional row in `ProductCategory` has no effect, because there is no row in `Product` with a matching `ProductCategoryId` column. This is where outer joins can come in handy. They enable you to include extra rows such as the one added to `ProductCategoryId`.

There are three types of outer join that enable you to specify which table in the join should have all of its rows included in the result set, regardless of the join specification. These are:

- ❑ **Left outer join:** Includes all rows in the first table specified in the result set.
- ❑ **Right outer join:** Includes all rows in the second table specified in the result set.
- ❑ **Full outer join:** Includes all rows in the both tables specified in the result set.

A left outer join is specified using the keywords `LEFT OUTER JOIN`:

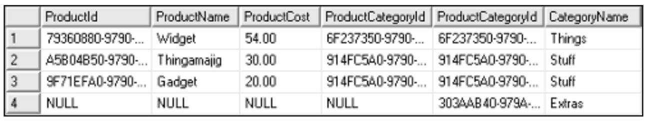
```
SELECT * FROM Product LEFT OUTER JOIN ProductCategory
   ON Product.ProductCategoryId = ProductCategory.ProductCategoryId
```

This specifies that all rows in `Product` should be included. However, looking at the result set in Figure 1-8, you can see that all the `Product` rows are already in the result. This query gives the same result as an inner join in this instance. This is, in fact, enforced by the foreign key relationship between the `Product` and `ProductCategory` tables. The `Product.ProductCategoryId` column cannot be null and cannot refer to a nonexistent row in `ProductCategory`, so all the rows in `Product` are, by implication, included in the result set.

A right outer join uses the keywords `RIGHT OUTER JOIN`:

```
SELECT * FROM Product LEFT OUTER JOIN ProductCategory
   ON Product.ProductCategoryId = ProductCategory.ProductCategoryId
```

This time, the result set is different, as shown in Figure 1-11.



	ProductId	ProductName	ProductCost	ProductCategoryId	ProductCategoryId	CategoryName
1	79360880-9790-...	Widget	54.00	6F237350-9790-...	6F237350-9790-...	Things
2	A5B04B50-9790-...	Thingamajig	30.00	914FC5A0-9790-...	914FC5A0-9790-...	Stuff
3	9F71EFA0-9790-...	Gadget	20.00	914FC5A0-9790-...	914FC5A0-9790-...	Stuff
4	NULL	NULL	NULL	NULL	303AAB40-979A-...	Extras

Figure 1-11: The result of a right outer join between Product and ProductCategory based on ProductCategoryId columns

The additional row (row 4 in Figure 1-11) includes the data from the new column in `ProductCategory` combined with a bunch of null values in the columns taken from `Product`.

A full outer join's keywords are, predictably, `FULL OUTER JOIN`:

```
SELECT * FROM Product FULL OUTER JOIN ProductCategory
   ON Product.ProductCategoryId = ProductCategory.ProductCategoryId
```

This gives the same result as a right outer join. It actually includes non-joined rows from both tables, but because this applies only to `ProductCategory` it is the same as a right outer join in this instance. That is not always the case.

Outer joins of all kinds are less commonly used than inner joins, but can still be useful. As a simple example, only by performing an outer join can you tell that there is a product category called `Extras` in the preceding results. This has important ramifications in many situations and can address issues that could not be solved using, for instance, only inner joins.

Bear in mind that this discussion has barely scratched the surface of what is possible with select queries. Select statements are capable of ordering data, limiting the total number of rows returned, grouping data and performing statistical analysis on these groups, calculating columns in the result set based on column data and the results of functions, renaming columns, and much more. Again, you'll see these features in action later in the book.

Adding Data

After the section on retrieving data, it may come as somewhat of a relief to learn that adding, deleting, and updating data are typically much simpler.

Adding data is achieved using the `INSERT` keyword. The basic syntax of an insert query that inserts a single row into a table is as follows:

```
INSERT INTO [Table] ([Column(s)]) VALUES ([Value(s)])
```

Here, `[Table]` is the table to insert into, `[Column(s)]` is a comma-separated list of columns to insert data into, and `[Value(s)]` is a comma-separated list of values to insert into those columns. For example:

```
INSERT INTO ProductCategory (ProductCategoryId, CategoryName)
   VALUES ('3bd514c0-97a1-11da-a72b-0800200c9a66', 'Doodads')
```

This may be simple, but there are several points to note:

- The `INTO` keyword is optional, but it makes the query easier to read. If you want, however, you can omit it from your queries.
- The list of columns specified may be a subset of the columns defined in the table for several reasons:
  - Some columns may allow null values, in which case adding data to them is optional.
  - Some columns may be defined with default values, in which case the default will be used if the column is omitted from the insert statement.
  - A column may be defined as an identity column, in which case the RDBMS may not allow you to insert data into it, and will generate an error if you attempt to do so. This is because identity columns are maintained by the RDBMS, and will be assigned values automatically.
- The columns don't have to be specified in the same order as they are defined in the table. The important thing to remember is that the column order specified in the insert statement *must* match the order of the values specified in the statement.
- Other rules applying to columns also apply, such as the rule that primary key values must be unique. The preceding query will execute fine once, but attempting to execute it again will result in an error because you will be attempting to insert duplicate values into `ProductCategoryId`.
- Null values can be inserted into columns using the keyword `NULL`.

For example, the following would be legal syntax for adding a row to the `PhoneBookEntry` described earlier in this chapter:

```
INSERT INTO PhoneBookEntry (EntryName, PhoneNumber, Address, IsIndividual)
VALUES ('Wayne Rooney', '555 123456', 'c/o Sven', 1)
```

The following would cause an error:

```
INSERT INTO PhoneBookEntry (EntryName, PhoneNumber, Address, IsIndividual)
VALUES ('Wayne Rooney', '555 123456', 1, 'c/o Sven')
```

In this example, the order of values supplied is wrong — it doesn't match the order specified by the column names.

This is the ideal place for a handy tip, and also the first time you will see a function in action. When using the `uniqueidentifier` type in SQL Server, which, as you have already seen, is a GUID value, you can use the `NEWID()` function to generate a GUID value. For example:

```
INSERT INTO ProductCategory (ProductCategoryId, CategoryName)
   VALUES (NEWID(), 'Doodads')
```

When this statement is executed, the `NEWID()` function is called and returns a GUID value that is used in the new row — meaning that you don't have to generate a GUID externally. Another practical function when adding rows is `GETDATE()`, which obtains the current date and time — useful when date-stamping rows. In SQL Server, you can use the `datetime` column data type for this purpose.

The INSERT keyword can also be used to insert multiple columns into a table at the same time. However, this cannot be achieved by specifying all of the column values as part of the statement. Instead, you must specify column values indirectly by using a SQL statement that returns a set of data — for example, a select query.

In insert statements of this form you must omit the VALUES keyword. The following is an example of a multirow insert query:

```
INSERT INTO ProductW (ProductId, ProductName, ProductCost, ProductCategoryId)
   SELECT ProductId, ProductName, ProductCost, ProductCategoryId FROM Product
   WHERE ProductName LIKE 'w%'
```

This statement copies all the products from the Product table whose ProductName column value starts with the letter w into a table called ProductW — which has exactly the same column specification as Product.

The query used to obtain data for inserting multiple rows needn't be this simple. The data could come from multiple tables, and the source columns needn't have the same names as the destination columns as long as their types match.

As with select statements, there is a lot more that you can do with insert statements, but these are the basics that you need to add data to database tables using SQL.

Deleting Data

You can delete data from databases using the DELETE keyword. But first, a word of warning — the delete statement makes it easy to accidentally delete the entire contents of a database table.

The syntax for a delete statement is as follows:

```
DELETE FROM [Table] WHERE [Filter]
```

Here, [Table] is the table from which to delete data, and [Filter] is a filter used to identify the data to delete. Delete statements operate on whole rows of data — it is not possible to delete individual columns from rows. The FROM keyword is optional (like the INTO keyword in insert statements, it can be more readable to leave it in), and the where clause is also optional.

If the where clause is omitted, *all* the rows in the table will be deleted. If you want this to happen, fine. If not, then be careful!

The following statement deletes all rows from the Product table:

```
DELETE FROM Product
```

As you can no doubt tell, this is a common mistake to make, and a serious one.

Using a filter, however, means that you can delete single records or a lot of records at once, depending on what you want to do. For example:

```
DELETE FROM ProductA WHERE ProductName NOT LIKE 'a%'
```

This query would delete all the rows from a table called ProductA that didn't have ProductName values that started with the letter A. Here's another example:

```
DELETE FROM ProductCategory
   WHERE ProductCategoryId = '3bd514c0-97a1-11da-a72b-0800200c9a66'
```

Because ProductCategory.ProductCategoryId is a primary key column that doesn't allow duplicate values, this command will delete zero or one row from the ProductCategory table, where the row that will be deleted has a ProductCategoryId column containing the GUID 3bd514c0-97a1-11da-a72b-0800200c9a66.

Updating Data

One way to update data in a database table is to delete a row and then add it again with slightly different data. However, that may be difficult or perhaps impossible to do if, for example, the table includes an identity column and you were required to keep the value of that column constant. Removing and then adding a row might also break relationships between rows, and the RDBMS may be configured to prevent you from doing this. In addition, this could cause conflicts and/or errors where multiple users access the database simultaneously.

Because of all this, the SQL specification includes another useful keyword to update data in existing rows: UPDATE. The syntax of an update statement is as follows:

```
UPDATE [Table] SET [Column Modification(s)] WHERE [Filter]
```

[Table] is the table containing the rows that you want to modify, [Column Modification(s)] is one or more comma-separated modifications to the rows in the table, and [Filter] filters the rows in the table that the update should apply to. As with previous queries, the where clause is optional.

Each column modification specification takes the following form:

```
[Column] = [Value]
```

[Column] is the name of the column to modify and [Value] is the value to replace the existing values in that column with. The value specified may be a simple literal value, or it may involve a calculation. If using a calculation, you can include the current value of a column in that calculation. For example:

```
UPDATE Product SET ProductCost = ProductCost * 1.1
```

This query would have the effect of increasing the cost of all products in the Product table by 10 percent, using the standard mathematical multiplication operator *.

As with delete queries, judicious use of the where clause may be required to restrict the rows where modifications should take place. Also, specifying a value for the primary key of a row in the where clause makes it possible to edit the content of individual rows.

Manipulating Databases

As well as being able to manipulate the data within databases, the SQL language includes all the commands you might need to manipulate database objects, including databases, stored procedures, tables, and so on.

For example, the following command, `CREATE DATABASE`, would create a new database within the DBMS:

```
CREATE DATABASE MyDatabaseOfWonders
```

Once created, you can add tables using additional SQL statements, although first you need to specify the name of the database where the statements will execute. To do so, you use the `USE` command:

```
USE MyDatabaseOfWonders
```

Then you can use a `CREATE TABLE` statement to add a table to your database:

```
CREATE TABLE [dbo].[Product]
(
   [ProductId] [uniqueidentifier] NOT NULL,
   [ProductName] [varchar](200) COLLATE Latin1_General_CI_AI NOT NULL,
   [ProductCost] [money] NOT NULL,
   [ProductCategoryId] [uniqueidentifier] NOT NULL,
   CONSTRAINT [PK_Product] PRIMARY KEY CLUSTERED
   (
      [ProductId] ASC
   ) ON [PRIMARY]
) ON [PRIMARY]
```

This command creates the `Product` table you've been looking at throughout this chapter. Some of the syntax used here is a little strange at first glance, but it's all easy enough to understand.

First, the table name is specified as `[dbo].[Product]`. This says that the `Product` table should belong to the `dbo` schema, where dbo is an abbreviation of database owner, and is a schema that exists in SQL Server 2005 databases by default. This additional specification is optional, and typically the `dbo` schema will be the default schema used unless additional configuration has taken place.

The next few lines specify the columns that will be contained by the table, by way of column names, data types, and whether they allow null values (the qualifier `NOT NULL` is used for rows that don't). Also, in the case of text fields, the collation is specified via the `COLLATE` keyword. The collation defines the character set to use, and therefore how the data will be stored in the database. (Different character sets require different amounts of storage for each character.)

After the column specifications, a constraint is added. Basically, constraints are additional properties that are applied to columns and define what values are allowed in columns, how the column data should be used (including key specifications), and indexing information. In this example, the `ProductId` column is made the primary key of the table with an ascending index and the key name `PK_Product`.

The final lines of code determine the partition that the table should exist in — in this case `PRIMARY`, which is the default installation of SQL Server.

One thing is missing here — there is no foreign key specification. Assuming that you had added the `ProductCategory` table, this specification would require a second command. However, before that second command runs, you need to make sure that the `CREATE TABLE` statement executes. To pause until previous statements have completed, use the simple SQL keyword `GO`:

```
GO
```

Then you would add the foreign key, in the form of another constraint:

```
ALTER TABLE [dbo].[Product] WITH CHECK ADD CONSTRAINT [FK_Product_ProductCategory]
   FOREIGN KEY ([ProductCategoryId])
   REFERENCES [dbo].[ProductCategory] ([ProductCategoryId])
GO
```

Here the `FK_Product_ProductCategory` foreign key is added, linking the `Product.ProductCategoryId` column with the `ProductCategory.ProductCategoryId` column.

There are several `CREATE` statements in the SQL vocabulary, each of which has a corresponding `ALTER` statement and also a corresponding `DROP` statement. Dropping an object means deleting it from the DBMS. This operation doesn't use the `DELETE` keyword, which is used for deleting data; mixing up these commands is potentially disastrous.

Although this chapter introduces a number of commands, you are far more likely to carry out these operations via a GUI for day-to-day use. That's fine, because making even simple mistakes with queries of this sort can cause irreparable damage.

The most useful thing about these statements so far is that they can be combined together into script files. Script files can be extremely useful for automating lengthy tasks, and the first place you will see one of these in action is in the next chapter. You will execute a SQL script file that creates the database that you will be using for examples throughout the book. The script file contains a complete database, including multiple tables, table data, and other database objects. If you were to add all of this by hand it would take a long time indeed, but executing a script takes hardly any time at all.

XML

As noted earlier, XML is a text-based format for the storage of data. It consists of two features: data and markup. Because it is "just" text, it can be read and understood on just about any computer system in existence, and its well-defined format makes it easy to process. It is also possible to define vocabularies — that is, systems of markup unique to an application or shared among many applications. As such, XML has become the universal language of choice for the interchange of information between disparate systems.

In the .NET Framework, XML is used extensively. For example, configuration files for all manner of applications are written in XML, which makes it easy to edit configurations by hand or programmatically using simple techniques. A rich set of types is defined by .NET to make it easy to manipulate XML data in various ways. XML is also the basis of SOAP (Simple Object Access Protocol), the underlying technology that makes web services both possible and platform-independent.

The rise of XML has continued unabatedly for several years, and knowledge of XML is now an essential part of pretty much any area of computing. This applies to databases, too, and many DBMSes (including SQL Server) now include tools for dealing with, consuming, and generating XML data.

The XML Format

This section cannot detail every aspect of the XML syntax, but it is a brief summary to reinforce the basic concepts of XML. After all, the chances are that you will have come across XML before, and if you haven't, there are a huge number of excellent resources, in web and print form, to get you started.

The markup in XML documents consists of data enclosed in elements, where that data may consist of nested (child) elements. Every XML document contains a single root (or document) element, and elements nested within the root element make up a hierarchy of data. Apart from the root element, XML documents may contain a single XML declaration, and zero or more processor directives. Each element, including the root element, has either a start tag and an end tag, or a single empty element tag. Start tags and empty element tags can include attributes that consist of name/value pairs.

Here's an example of an XML document:

```
<?xml version="1.0" encoding="utf-8" ?>
<foodStuffs>
  <foodStuff category="pizza">
    <name>Cheese and Tomato</name>
    <size>10"</size>
    <rating>4*</rating>
  </foodStuff>
  <foodStuff category="pizza">
    <name>Four Seasons</name>
    <size>8"</size>
    <rating>1*</rating>
    <isNasty />
  </foodStuff>
</foodStuffs>
```

The XML declaration on the first line of the document identifies the version of XML to which the document conforms and how it is encoded (that is, the character set used). The root element of the document is `<foodStuffs>`, which contains two `<foodStuff>` elements. Each `<foodStuff>` element has an attribute called `category`, and child elements called `<name>`, `<size>`, and `<rating>`, each of which contains text data. Each of these elements consists of a start tag (for example, `<size>`) and an end tag that includes a preceding front slash (`</size>`). The second `<foodStuff>` element also contains an empty element, `<isNasty>`, which includes a trailing front slash to indicate that the element is empty. One way of looking at this document is as an array of `foodStuff` objects, each of which has properties that are represented as attributes or child elements.

There are a few more rules concerning XML documents. For a start, they are case-sensitive. `<foodStuff>` and `<Foodstuff>`, for example, are interpreted as two completely different elements. Also, every start tag must have a matching end tag, and elements can't overlap, that is to say that the following is illegal:

```
<element1><element2></element1></element2>
```

Here's an example in which `<element2>` is correctly nested inside `<element1>`:

```
<element1><element2></element2></element1>
```

And in this example, neither element is nested in the other:

```
<element1></element1>
<element2></element2>
```

Storing XML Data

In the XML document shown in the previous section, it is fairly obvious how the data might map to a database table. For instance, you could have a `FoodStuff` table containing columns for `Category` (possible a foreign key field linking to a separate `Category` table), text columns for `Name`, `Size`, and `Rating`, and a `bit` type column for `IsNasty`. In the XML, the root `<foodStuffs>` element would simply be used as a placeholder containing the data, and each `<foodStuff>` element would represent a row in the table.

However, XML documents can come in other forms, too. Here's an example:

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
  <h1>Funny bone results in dog days!</h1>
  Rumor has it <i>(sources unknown)</i> that a well known <br />
  comedy dog double act is due to split any day now.<br />
  <br />
  The latest information to come out of the rumor mill is that<br />
  a dispute arose about the location of a <b>buried bone</b>, and that<br />
  until it is found the dogs in question are only communicating<br />
  via their lawyers.<br />
  <br />
  More news as it happens!
</body>
```

This is, in fact, a piece of HTML. But it's a little more than that — it's actually a fragment of XHTML — an XML dialect of HTML. It's also a perfectly legal XML document, but creating a table capable of holding this information in row form would be practically impossible. Instead, storing this in a database would mean putting the whole lot in a single column of a row, in text form. SQL Server includes an `xml` datatype for storing this sort of data, or you could just use a text column. When you store data using the `xml` datatype, however, there is additional functionality that you can use, such as querying data within the document using the XQuery language.

You look at this facet of SQL Server later in the book.

Retrieving Data as XML

As well as being able to retrieve XML data stored in `xml` type columns directly as XML, SQL Server also makes it possible to retrieve data from any result set in the form of XML data. This involves an additional `FOR XML` clause, which has a number of uses and ways of customizing the format in which XML data is obtained, but can also be used simply. For example:

```
SELECT * FROM Product FOR XML AUTO
```

This query obtains data as a single string as follows:

```
<Product ProductId="A5B04B50-9790-11DA-A72B-0800200C9A66"
  ProductName="Thingamajig" ProductCost="30.0000"
  ProductCategoryId="914FC5A0-9790-11DA-A72B-0800200C9A66"/>
<Product ProductId="79360880-9790-11DA-A72B-0800200C9A66"
  ProductName="Widget" ProductCost="54.0000"
  ProductCategoryId="6F237350-9790-11DA-A72B-0800200C9A66"/>
```

```
   <Product ProductId="9F71EFA0-9790-11DA-A72B-0800200C9A66"
     ProductName="Gadget" ProductCost="20.0000"
     ProductCategoryId="914FC5A0-9790-11DA-A72B-0800200C9A66"/>
```

This is not a complete, legal XML document as it stands (it has multiple root elements for one thing), but it would be easy to turn it into one.

If you are writing applications that must generate XML from data stored in a database, the `FOR XML` clause can speed things up dramatically — because by using the right queries it would be possible to avoid having to do any further data processing outside of SQL Server.

SQL Server also provides ways to insert rows into tables directly from XML documents and even has the capability to return XML data in response to web requests. Again, these are things that you will see later in the book, as they become pertinent.

Summary

In this chapter, you looked at the basics of databases, including how they are structured and how to access the data contained in them. You have also learned about the additional features of databases and how to use SQL to manipulate databases, and you saw a quick summary of XML and how it fits in to the database world.

Specifically, you have learned:

- What a database is
- What terminology to use when referring to databases
- How relational databases work, and what makes them useful
- What the difference is between relational and object-oriented database management systems
- What functionality databases offer above and beyond storing data
- What the differences are between many of the available DBMSes
- What SQL is
- How to retrieve, add, delete, and update data in databases using a variety of SQL queries
- What else is possible using more advanced SQL syntax
- What XML is
- How it is possible to use XML data in combination with databases

In the next chapter, you see how C# can be used to interact with SQL Server 2005 Express Edition, and you start to experiment with sample applications.

Exercises

1. Database tables must include primary keys. Is this statement true or false?
2. Which of the following are actual types of joins between tables?
   a. Inner joins
   b. Sideways joins
   c. Internal joins
   d. Left outer joins
   e. Dovetail joins
3. If you wanted to perform two update queries in which either both queries must succeed or both must fail, what technology would you use?
4. What is wrong with the following SQL statements?

```
DELETE FROM MyTable
UPDATE MyTable (Title, Amount) SET ('Oysters', 17) WHERE ItemId = 3
```

5. Any XML document may be inserted into a SQL database table as a set of rows. Is this statement true or false? Why?