1

RDBMS Basics: What Makes Up a SQL Ser ver Database?

What makes up a database? Data for sure. (What use is a database that doesn't store anything?) But a *Relational Database Management System* (*RDBMS*) is actually much more than data. Today's advanced RDBMSs not only store your data, they also manage that data for you, restricting the kind of data that can go into the system, and facilitating getting data out of the system. If all you want is to tuck the data away somewhere safe, you could use just about any data storage system. RDBMSs allow you to go beyond the storage of the data into the realm of defining what that data should look like, or the *business rules* of the data.

Don't confuse what I'm calling the "business rules of data" with the more generalized business rules that drive your entire system (for example, preventing someone from seeing anything until they've logged in, or automatically adjusting the current period in an accounting system on the first of the month). Those types of rules can be enforced at virtually any level of the system (these days, it's usually in the middle or client tier of an n-tier system). Instead, what we're talking about here are the business rules that specifically relate to the data. For example, you can't have a sales order with a negative amount. With an RDBMS, we can incorporate these rules right into the integrity of the database itself.

The notion of the database taking responsibility for the data within, as well as the best methods to input and extract data from that database, serve as the foundation of what this book is all about. This chapter provides an overview of the rest of the book. Most items discussed in this chapter are covered again in later chapters, but this chapter is intended to provide you with a road map or plan to bear in mind as we progress through the book. With this in mind, we'll take a high-level look into:

- Database objects
- Data types
- Other database concepts that ensure data integrity

An Overview of Database Objects

An instance of an RDBMS such as SQL Server contains many *objects*. Object purists out there may quibble with whether Microsoft's choice of what to call an object (and what not to) actually meets the normal definition of an object, but, for SQL Server's purposes, the list of some of the more important database objects can be said to contain such things as:

- The database itself
- The transaction log
- Indexes
- Filegroups
- Diagrams
- Views
- Stored procedures
- User-defined functions
- Users
- Roles
- Assemblies
- Tables
- Reports
- Full-text catalogs
- User-defined data types

The Database Object

The database is effectively the highest-level object that you can refer to within a given SQL Server. (Technically speaking, the server itself can be considered to be an object, but not from any real "programming" perspective, so we're not going there.) Most, but not all, other objects in a SQL Server are children of the database object.

> *If you are already familiar with SQL Server you may now be saying, "What? What happened to logins or SQL Agent tasks?" SQL Server has several other objects (as listed previously) that exist in support of the database. With the exception of linked servers, and perhaps Integration Services packages, these are primarily the domain of the database administrator and, as such, we generally don't give them significant thought during the design and programming processes. (They are programmable via something called the SQL Management Objects (SMO), which is beyond the scope of this book.) While there are some exceptions to this rule, I generally consider them to be advanced in nature, and thus save them for the Professional version of this book.*

A database is typically a group of constructs that include at least a set of table objects and, more often than not, other objects, such as stored procedures and views that pertain to the particular grouping of data stored in the database's tables.

What types of tables do we store in just one database and what goes in a separate database? We discuss that in some detail later in the book, but for now we'll take the simple approach of saying that any data that is generally thought of as belonging to just one system, or is significantly related, will be stored in a single database. An RDBMS, such as SQL Server, may have multiple databases on just one server, or it may have only one. The number of databases that reside on an individual SQL Server depends on such factors as capacity (CPU power, disk I/O limitations, memory, and so on), autonomy (you want one

person to have management rights to the server this system is running on, and someone else to have admin rights to a different server), and just how many databases your company or client has. Some servers have only one production database; others may have many. Also, keep in mind that with any version of SQL Server that you're likely to find in production these days (SQL Server 2000 was already five years old by the time it was replaced, so we'll assume most shops have that or higher), we have the ability to have multiple instances of SQL Server — complete with separate logins and management rights — all on the same physical server.

> *I'm sure many of you are now asking, "Can I have different versions of SQL Server on the same box — say, SQL Server 2005 and SQL Server 2008?" The answer is yes. You can mix SQL Server 2005 and 2008 on the same box. Personally, I am not at all trusting of this configuration, even for migration scenarios, but, if you have the need, yes, it can be done.*

When you first load SQL Server, you start with at least four system databases:

- `master`
- `model`
- `msdb`
- `tempdb`

All of these need to be installed for your server to run properly. (Indeed, without some of them, it won't run at all.) From there, things vary depending on which installation choices you made. Examples of some of the databases you may see include the following:

- ReportServer (the database that serves Reporting Server configuration and model storage needs)
- ReportServerTempDB (the working database for Reporting Server)
- AdventureWorks2008 (the sample database)
- AdventureWorksLT2008 (a new, "lite" version of the sample database)
- AdventureWorksDW2008 (sample for use with Analysis Services)

In addition to the system-installed examples, you may, when searching the Web or using other tutorials, find reference to a couple of older samples:

- `pubs`
- `Northwind`

> *In the previous edition of this book, I made extensive use of the older examples. Unfortunately, there is little guarantee how long those examples will remain downloadable, and, as such, I made the choice to switch over to the new examples. The newer AdventureWorks2008 database is certainly a much more robust example and does a great job of providing examples of just about every little twist and turn you can make use of in SQL Server 2008. There is, however, a problem with that — complexity. The Adventure-Works2008 database is excessively complex for a training database. It takes features that are likely to be used only in exceptional cases and uses them as a dominant feature. So, with that said, let me make the point now that AdventureWorks2008 should not necessarily be used as a template for what to do in other similar applications.*

The master Database

Every SQL Server, regardless of version or custom modifications, has the `master` database. This database holds a special set of tables (system tables) that keeps track of the system as a whole. For example, when you create a new database on the server, an entry is placed in the `sysdatabases` table in the master database. All extended and system-stored procedures, regardless of which database they are intended for use with, are stored in this database. Obviously, since almost everything that describes your server is stored in here, this database is critical to your system and cannot be deleted.

The system tables, including those found in the master database, were, in the past, occasionally used in a pinch to provide system configuration information, such as whether certain objects existed before you performed operations on them. Microsoft warned us for years not to use the system tables directly, but, since we had few other options, most developers ignored that advice. Happily, Microsoft began giving us other options in the form of system and information schema views; we can now utilize these views to get at our systems' metadata as necessary with Microsoft's full blessing. For example, if you try to create an object that already exists in any particular database, you get an error. If you want to force the issue, you could test to see whether the table already has an entry in the `sys.objects` table for that database. If it does, you would delete that object before re-creating it.

> **If you're quite cavalier, you may be saying to yourself, "Cool, I can't wait to mess around in those system tables!"** ***Don't go there!*** **Using the system tables in any form is fraught with peril. Microsoft makes absolutely no guarantees about compatibility in the master database between versions. Indeed, they virtually guarantee that they will change. Fortunately, several alternatives (for example, system functions, system stored procedures, and `information_schema` views) are available for retrieving much of the metadata that is stored in the system tables.**
>
> **All that said, there are still times when nothing else will do, but, in general, you should consider them to be evil cannibals from another tribe and best left alone.**

The model Database

The `model` database is aptly named, in the sense that it's the model on which a copy can be based. The model database forms a template for any new database that you create. This means that you can, if you wish, alter the `model` database if you want to change what standard, newly created databases look like. For example, you could add a set of audit tables that you include in every database you build. You could also include a few user groups that would be cloned into every new database that was created on the system. Note that since this database serves as the template for any other database, it's a required database and must be left on the system; you cannot delete it.

There are several things to keep in mind when altering the model database. First, any database you create has to be at least as large as the model database. That means that if you alter the `model` database to be 100MB in size, you can't create a database smaller than 100MB. There are several other similar pitfalls. As such, for 90 percent of installations, I strongly recommend leaving this one alone.

The msdb Database

`msdb` is where the SQL Agent process stores any system tasks. If you schedule backups to run on a database nightly, there is an entry in `msdb`. Schedule a stored procedure for one-time execution, and yes, it

has an entry in msdb. Other major subsystems in SQL Server make similar use of `msdb`. SSIS packages and policy-based management definitions are examples of other processes that make use of msdb.

The tempdb Database

`tempdb` is one of the key working areas for your server. Whenever you issue a complex or large query that SQL Server needs to build interim tables to solve, it does so in tempdb. Whenever you create a temporary table of your own, it is created in tempdb, even though you think you're creating it in the current database. (An alias is created in the local database for you to reference it by, but the physical table is created in tempdb). Whenever there is a need for data to be stored temporarily, it's probably stored in tempdb.

tempdb is very different from any other database. Not only are the objects within it temporary, the database itself is temporary. It has the distinction of being the only database in your system that is completely rebuilt from scratch every time you start your SQL Server.

> **Technically speaking, you can actually create objects yourself in tempdb. I strongly recommend against this practice. You can create temporary objects from within any database you have access to in your system — they will be stored in tempdb. Creating objects directly in tempdb gains you nothing, but adds the confusion of referring to things across databases. This is another of those "Don't go there!" kind of things.**

ReportServer

This database will only exist if you installed ReportServer. (It does not necessarily have to be the same server as the database engine, but note that, if it is a different server, then it requires a separate license.) The ReportServer database stores any persistent metadata for your Reporting Server instance. Note that this is purely an operational database for a given Reporting Server instance, and should not be modified or accessed other than through the Reporting Server.

ReportServerTempDB

This serves the same basic function as the ReportServer database, except that it stores nonpersistent data (such as working data for a report that is running). Again, this is a purely operational database, and you should not access or alter it in any way except through the Reporting Server.

AdventureWorks2008

SQL Server included samples long before this one came along. The old samples had their shortcomings, though. For example, they contained a few poor design practices. In addition, they were simplistic and focused on demonstrating certain database concepts rather than on SQL Server as a product, or even databases as a whole. I'll hold off the argument of whether AdventureWorks2008 has the same issues or not. Let's just say that AdventureWorks2008 was, among other things, an attempt to address this problem.

From the earliest stages of development of SQL Server 2005, Microsoft knew they wanted a far more robust sample database that would act as a sample for as much of the product as possible. AdventureWorks2008 is the outcome of that effort. As much as you will hear me complain about its overly complex nature for the beginning user, it is a masterpiece in that it shows it *all* off. Okay, so it's not really *everything*, but it is a fairly complete sample, with more realistic volumes of data, complex structures, and sections that show samples for the vast majority of product features. In this sense, it's truly terrific.

AdventureWorks2008 will be something of our home database — we use it extensively as we work through the examples in this book.

AdventureWorksLT2008

The "LT" in this stands for lite. This is just an extremely small subset of the full AdventureWorks2008 database. The idea is to provide a simpler sample set for easier training of basic concepts and simple training. While I've not been privy to the exact reasoning behind this new sample set, my suspicion is that it is an effect to try and kill the older, Northwind and pubs sample sets, which have been preferred by many trainers over the newer AdventureWorks2008 set, as the AdventureWorks2008 database is often far too complex and cumbersome for early training.

AdventureWorksDW2008

This is the Analysis Services sample. (The DW stands for Data Warehouse, which is the type of database over which most Analysis Services projects will be built.) Perhaps the greatest thing about it is that Microsoft had the foresight to tie the transaction database sample with the analysis sample, providing a whole set of samples that show the two of them working together.

Decision support databases are well outside the scope of this book, and you won't be using this database, but keep it in mind as you fire up Analysis Services and play around. Take a look at the differences between the two databases. They are meant to serve the same fictional company, but they have different purposes: Learn from it.

The pubs Database

Ahhhh, pubs! It's almost like an old friend. pubs is one of the original example databases and was supplied with SQL Server as part of the install prior to SQL Server 2005. It is now only available as a separate download from the Microsoft Website. You will still find many training articles and books that refer to pubs, but Microsoft has made no promises regarding how long they will continue to make it available. pubs has absolutely nothing to do with the operation of SQL Server. It is merely there to provide a consistent place for your training and experimentation. You do not need pubs to work the examples in this book, but you may want to download and install it to work with other examples and tutorials you may find on the Web.

The Northwind Database

If your past programming experience has involved Access or Visual Basic, then you should already be somewhat familiar with the Northwind database. Northwind was added to SQL Server beginning in version 7.0, but was removed from the basic installation as of SQL Server 2005. Much like pubs, it can, for now, be downloaded separately from the base SQL Server install. (Fortunately, it is part of the same sample download and install as pubs is.) Like pubs, you do not need the Northwind database to work the examples in this book, but it is handy to have available for work with various examples and tutorials you will find on the Web.

The Transaction Log

Believe it or not, the database file itself isn't where most things happen. Although the data is certainly read in from there, any changes you make don't initially go to the database itself. Instead, they are written serially to the *transaction log*. At some later point in time, the database is issued a *checkpoint*; it is at that point in time that all the changes in the log are propagated to the actual database file.

The database is in a random access arrangement, but the log is serial in nature. While the random nature of the database file allows for speedy access, the serial nature of the log allows things to be tracked in the proper order. The log accumulates changes that are deemed as having been committed, and then writes several of them at a time to the physical database file(s).

We'll take a much closer look at how things are logged in Chapter 14, but for now, remember that the log is the first place on disk that the data goes, and it's propagated to the actual database at a later time. You need both the database file and the transaction log to have a functional database.

The Most Basic Database Object: Table

Databases are made up of many things, but none is more central to the make-up of a database than tables are. A table can be thought of as equating to an accountant's ledger or an Excel spreadsheet. It is made up of what is called *domain* data (columns) and *entity* data (rows). The actual data for the database is stored in the tables.

Each table definition also contains the *metadata* (descriptive information about data) that describes the nature of the data it is to contain. Each column has its own set of rules about what can be stored in that column. A violation of the rules of any one column can cause the system to reject an inserted row, an update to an existing row, or the deletion of a row.

Let's take a look at the Production.Location table in the AdventureWorks2008 database. (The view presented in Figure 1-1 is from the SQL Server Management Studio. This is a fundamental tool and we will look at how to make use of it in the next chapter.)

	LocationID	Name	CostRate	Availability	ModifiedDate
1	1	Tool Crib	0.00	0.00	1998-06-01 00:00:00.000
2	2	Sheet Metal Racks	0.00	0.00	1998-06-01 00:00:00.000
3	3	Paint Shop	0.00	0.00	1998-06-01 00:00:00.000
4	4	Paint Storage	0.00	0.00	1998-06-01 00:00:00.000
5	5	Metal Storage	0.00	0.00	1998-06-01 00:00:00.000
6	6	Miscellaneous Storage	0.00	0.00	1998-06-01 00:00:00.000
7	7	Finished Goods Storage	0.00	0.00	1998-06-01 00:00:00.000
8	10	Frame Forming	22.50	96.00	1998-06-01 00:00:00.000
9	20	Frame Welding	25.00	108.00	1998-06-01 00:00:00.000
10	30	Debur and Polish	14.50	120.00	1998-06-01 00:00:00.000
11	40	Paint	15.75	120.00	1998-06-01 00:00:00.000
12	45	Specialized Paint	18.00	80.00	1998-06-01 00:00:00.000
13	50	Subassembly	12.25	120.00	1998-06-01 00:00:00.000
14	60	Final Assembly	12.25	120.00	1998-06-01 00:00:00.000

Figure 1-1

The table in Figure 1-1 is made up of five columns of data. The number of columns remains constant regardless of how much data (even zero) is in the table. Currently, the table has fourteen records. The number of records will go up and down as we add or delete data, but the nature of the data in each record (or row) is described and restricted by the *data type* of the column.

Indexes

An *index* is an object that exists only within the framework of a particular table or view. An index works much like the index does in the back of an encyclopedia. There is some sort of lookup (or "key") value

that is sorted in a particular way and, once you have that, you are provided another key with which you can look up the actual information you were after.

An index provides us ways of speeding the lookup of our information. Indexes fall into two categories:

- **Clustered** — You can have only one of these per table. If an index is clustered, it means that the table on which the clustered index is based is physically sorted according to that index. If you were indexing an encyclopedia, the clustered index would be the page numbers (the information in the encyclopedia is stored in the order of the page numbers).
- **Non-clustered** — You can have many of these for every table. This is more along the lines of what you probably think of when you hear the word "index." This kind of index points to some other value that will let you find the data. For our encyclopedia, this would be the keyword index at the back of the book.

Note that views that have indexes — or *indexed views* — must have at least one clustered index before they can have any non-clustered indexes.

Triggers

A *trigger* is an object that exists only within the framework of a table. Triggers are pieces of logical code that are automatically executed when certain things (such as inserts, updates, or deletes) happen to your table.

Triggers can be used for a great variety of things, but are mainly used for either copying data as it is entered, or checking the update to make sure that it meets some criteria.

Constraints

A *constraint* is yet another object that exists only within the confines of a table. Constraints are much like they sound; they confine the data in your table to meet certain conditions. Constraints, in a way, compete with triggers as possible solutions to data integrity issues. They are not, however, the same thing: Each has its own distinct advantages.

Filegroups

By default, all your tables and everything else about your database (except the log) are stored in a single file. That file is, by default, a member of what's called the *primary filegroup*. However, you are not stuck with this arrangement.

SQL Server allows you to define a little over 32,000 *secondary files*. (If you need more than that, perhaps it isn't SQL Server that has the problem.) These secondary files can be added to the primary filegroup or created as part of one or more *secondary filegroups*. While there is only one primary filegroup (and it is actually called "Primary"), you can have up to 255 secondary filegroups. A secondary filegroup is created as an option to a `CREATE DATABASE` or `ALTER DATABASE` command.

Diagrams

We will discuss database diagramming in some detail when we discuss normalization and database design. For now, suffice it to say that a database diagram is a visual representation of the database

design, including the various tables, the column names in each table, and the relationships between tables. In your travels as a developer, you may have heard of an *entity-relationship diagram (ERD)*. In an ERD the database is divided into two parts: entities (such as "supplier" and "product") and relations (such as "supplies" and "purchases").

> *The included database design tools are, unfortunately, a bit sparse. Indeed, the diagramming methodology the tools use does not adhere to any of the accepted standards in ER diagramming. Still, these diagramming tools really do provide all the "necessary" things, so they are at least something of a start.*

Figure 1-2 is a diagram that shows some of the various tables in the `AdventureWorks2008` database. The diagram also (though it may be a bit subtle since this is new to you) describes many other properties about the database. Notice the tiny icons for keys and the infinity sign. These depict the nature of the relationship between two tables. We'll talk about relationships extensively in Chapters 6 and 8, and we'll look further into diagrams later in the book.

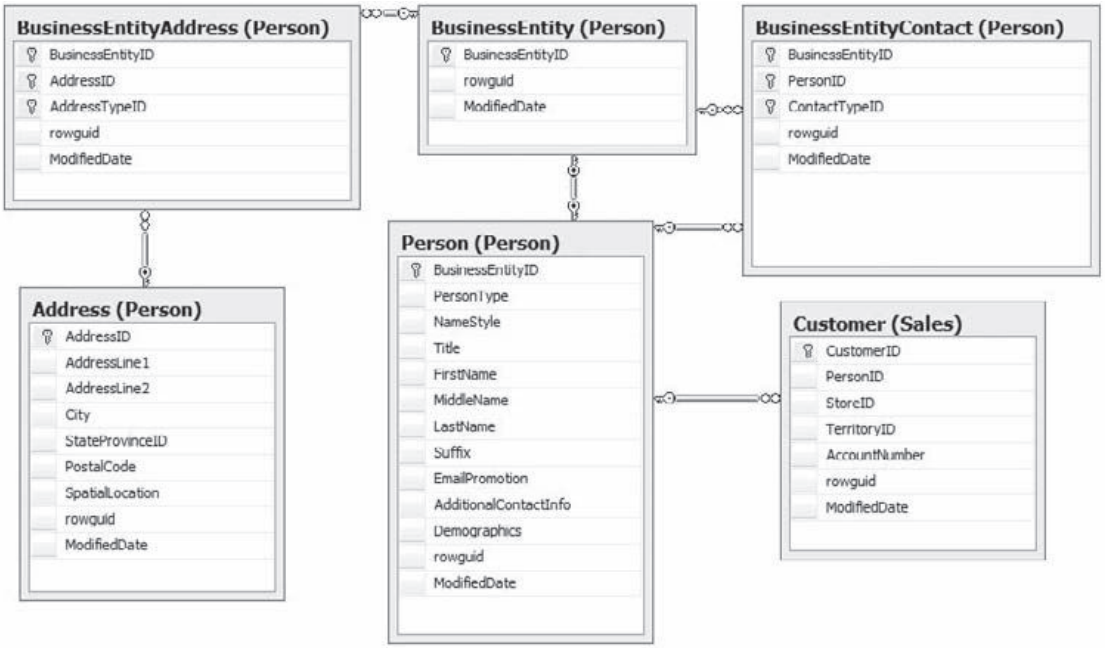

Figure 1-2

Views

A *view* is something of a virtual table. A view, for the most part, is used just like a table, except that it doesn't contain any data of its own. Instead, a view is merely a preplanned mapping and representation of the data stored in tables. The plan is stored in the database in the form of a query. This query calls for data from some, but not necessarily all, columns to be retrieved from one or more tables. The data retrieved may or may not (depending on the view definition) have to meet special criteria in order to be shown as data in that view.

Until SQL Server 2000, the primary purpose of views was to control what the user of the view saw. This has two major impacts: security and ease of use. With views you can control what the users see, so if

there is a section of a table that should be accessed by only a few users (for example, salary details), you can create a view that includes only those columns to which everyone is allowed access. In addition, the view can be tailored so that the user doesn't have to search through any unneeded information.

In addition to these most basic uses for views, you also have the ability to create what is called an *indexed view*. This is the same as any other view, except that you can now create an index against the view. This results in a couple of performance impacts (some positive, one negative):

- Views that reference multiple tables generally have *much* better read performance with an indexed view, because the join between the tables is preconstructed.
- Aggregations performed in the view are precalculated and stored as part of the index; again, this means that the aggregation is performed one time (when the row is inserted or updated), and then can be read directly from the index information.
- Inserts and deletes have higher overhead because the index on the view has to be updated immediately; updates also have higher overhead if the key column or the cluster key of the index is affected by the update.

We will look into these performance issues more deeply in Chapter 10.

Stored Procedures

Stored procedures (or *sprocs*) are the bread and butter of programmatic functionality in SQL Server. Stored procedures are generally an ordered series of Transact-SQL (the language used to query Microsoft SQL Server) statements bundled up into a single logical unit. They allow for variables and parameters, as well as selection and looping constructs. Sprocs offer several advantages over just sending individual statements to the server in the sense that they:

- Are referred to using short names, rather than a long string of text, therefore less network traffic is required in order to run the code within the sproc.
- Are pre-optimized and precompiled, saving a small amount of time each time the sproc is run.
- Encapsulate a process, usually for security reasons or just to hide the complexity of the database.
- Can be called from other sprocs, making them reusable in a somewhat limited sense.

While sprocs are the core of programmatic functionality in SQL Server, be careful in their use. They are often a solution, but they are also frequently not the only solution. Make sure they are the right choice before selecting a sproc as the option you go with.

User-Defined Functions

User-Defined Functions (UDFs) have a tremendous number of similarities to sprocs, except that they:

- Can return a value of most SQL Server data types. Excluded return types include `text`, `ntext`, `image`, `cursor`, and `timestamp`.
- Can't have side effects. Basically, they can't do anything that reaches outside the scope of the function, such as changing tables, sending e-mails, or making system or database parameter changes.

UDFs are similar to the functions that you would use in a standard programming language such as VB.NET or C++. You can pass more than one variable in, and get a value out. SQL Server's UDFs vary from the functions found in many procedural languages, in that *all* variables passed into the function are passed in by value. If you're familiar with passing in variables `By Ref` or passing in pointers, sorry, there is no equivalent here. There is, however, some good news in that you can return a special data type called a `table`. We'll examine the impact of this in Chapter 13.

Users and Roles

These two go hand in hand. *Users* are pretty much the equivalent of logins. In short, this object represents an identifier for someone to log in to the SQL Server. Anyone logging in to SQL Server has to map (directly or indirectly, depending on the security model in use) to a user. Users, in turn, belong to one or more *roles*. Rights to perform certain actions in SQL Server can then be granted directly to a user or to a role to which one or more users belong.

Rules

Rules and constraints provide restriction information about what can go into a table. If an updated or inserted record violates a rule, then that insertion or update will be rejected. In addition, a rule can be used to define a restriction on a *user-defined data type*. Unlike constraints, rules aren't bound to a particular table. Instead they are independent objects that can be bound to multiple tables or even to specific data types (which are, in turn, used in tables).

Rules have been considered deprecated by Microsoft for several releases now. They should be considered there for backward compatibility only and should be avoided in new development.

> *Given that Microsoft has introduced some new deprecation-management functionality in SQL Server 2008, I suspect that features (such as rules) that have been deprecated for several versions may finally be removed in the next version of SQL Server. As such, I feel the need to stress again that rules should not be utilized for new development. Indeed, it is probably long past time to actively migrate away from them.*

Defaults

There are two types of defaults. There is the default that is an object unto itself, and the default that is not really an object, but rather metadata describing a particular column in a table (in much the same way that we have rules, which are objects, and constraints, which are not objects, but metadata). They both serve the same purpose. If, when inserting a record, you don't provide the value of a column and that column has a default defined, a value will be inserted automatically as defined in the default. You will examine both types of defaults in Chapter 6.

User-Defined Data Types

User-defined data types are either extensions to the system-defined data types or complex data types defined by a method in a .NET assembly. The possibilities here are almost endless. Although SQL Server 2000 and earlier had the idea of user-defined data types, they were really limited to different filtering of existing data types. With releases since SQL Server 2005, you have the ability to bind .NET assemblies to your own data types, meaning you can have a data type that stores (within reason) about anything you

can store in a .NET object. Indeed, the new spatial data types (`Geographic` and `Geometric`) that have been added in SQL Server 2008 are implemented using a user-defined type based on a .NET assembly. .NET assemblies are covered in detail in *Professional SQL Server 2008 Programming*.

> *Careful with this! The data type that you're working with is pretty fundamental to your data and its storage. Although being able to define your own thing is very cool, recognize that it will almost certainly come with a large performance cost. Consider it carefully, be sure it's something you need, and then, as with everything like this, TEST, TEST, TEST!!!*

Full-Text Catalogs

Full-text catalogs are mappings of data that speed the search for specific blocks of text within columns that have full-text searching enabled. Prior to SQL Server 2008, full-text catalogs were stored external to the database (thus creating some significant backup and recovery issues). As of SQL Server 2008, full-text catalogs have been integrated into the mail database engine and storage mechanisms. Full text indexes are beyond the scope of this text, but are covered extensively in *Professional SQL Server 2008 Programming*.

SQL Server Data Types

Now that you're familiar with the base objects of a SQL Server database, let's take a look at the options that SQL Server has for one of the fundamental items of any environment that handles data — data types. Note that, since this book is intended for developers, and that no developer could survive for 60 seconds without an understanding of data types, I'm going to assume that you already know how data types work, and just need to know the particulars of SQL Server data types.

SQL Server 2008 has the intrinsic data types shown in the following table:

Data Type Name	Class	Size in Bytes	Nature of the Data
`Bit`	Integer	1	The size is somewhat misleading. The first `bit` data type in a table takes up 1 byte; the next 7 make use of the same byte. Allowing nulls causes an additional byte to be used.
`Bigint`	Integer	8	This just deals with the fact that we use larger and larger numbers on a more frequent basis. This one allows you to use whole numbers from -2^{63} to $2^{63}-1$. That's plus or minus about 92 quintrillion.
`Int`	Integer	4	Whole numbers from –2,147,483,648 to 2,147,483,647.

Data Type Name	Class	Size in Bytes	Nature of the Data
`SmallInt`	Integer	2	Whole numbers from –32,768 to 32,767.
`TinyInt`	Integer	1	Whole numbers from 0 to 255.
`Decimal` or `Numeric`	Decimal/ Numeric	Varies	Fixed precision and scale from $-10^{38}-1$ to $10^{38}-1$. The two names are synonymous.
`Money`	Money	8	Monetary units from -2^{63} to 2^{63} plus precision to four decimal places. Note that this could be any monetary unit, not just dollars.
`SmallMoney`	Money	4	Monetary units from –214,748.3648 to +214,748.3647.
`Float` (also a synonym for ANSI `Real`)	Approximate Numerics	Varies	Accepts an argument (for example, `Float(20)`) that determines size and precision. Note that the argument is in bits, not bytes. Ranges from –1.79E + 308 to 1.79E + 308.
`DateTime`	Date/Time	8	Date and time data from January 1, 1753, to December 31, 9999, with an accuracy of three hundredths of a second.
`DateTime2`	Date/Time	Varies (6–8)	Updated incarnation of the more venerable DateTime data type. Supports larger date ranges and large time-fraction precision (up to 100 nanoseconds). Like DateTime, it is not time zone aware, but does align with the .NET DateTime data type.
`SmallDateTime`	Date/Time	4	Date and time data from January 1, 1900, to June 6, 2079, with an accuracy of one minute.
`DateTimeOffset`	Date/Time	Varies (8–10)	Similar to the DateTime data type, but also expects an offset designation of –14:00 to +14:00 offset from UTC time. Time is stored internally as UTC time, and any comparisons, sorts, or indexing will be based on that unified time zone.

Continued

Data Type Name	Class	Size in Bytes	Nature of the Data
`Date`	Date/Time	3	Stores only date data from January 1, 0001, to December 31, 9999, as defined by the Gregorian calendar. Assumes the ANSI standard date format (YYYY-MM-DD), but will implicitly convert from several other formats.
`Time`	Date/Time	Varies (3–5)	Stores only time data in user-selectable precisions as granular as 100 nanoeconds (which is the default).
`Cursor`	Special Numeric	1	Pointer to a cursor. While the pointer takes up only a byte, keep in mind that the result set that makes up the actual cursor also takes up memory. Exactly how much will vary depending on the result set.
`Timestamp/ rowversion`	Special Numeric (binary)	8	Special value that is unique within a given database. Value is set by the database itself automatically every time the record is either inserted or updated, even though the timestamp column wasn't referred to by the `UPDATE` statement (you're actually not allowed to update the timestamp field directly).
`UniqueIdentifier`	Special Numeric (binary)	16	Special Globally Unique Identifier (GUID) is guaranteed to be unique across space and time.
`Char`	Character	Varies	Fixed-length character data. Values shorter than the set length are padded with spaces to the set length. Data is non-Unicode. Maximum specified length is 8,000 characters.
`VarChar`	Character	Varies	Variable-length character data. Values are not padded with spaces. Data is non-Unicode. Maximum specified length is 8,000 characters, but you can use the "max" keyword to indicate it as essentially a very large character field (up to 2^31 bytes of data).
`Text`	Character	Varies	Legacy support as of SQL Server 2005. Use `varchar(max)` instead!

Data Type Name	Class	Size in Bytes	Nature of the Data
`NChar`	Unicode	Varies	Fixed-length Unicode character data. Values shorter than the set length are padded with spaces. Maximum specified length is 4,000 characters.
`NVarChar`	Unicode	Varies	Variable-length Unicode character data. Values are not padded. Maximum specified length is 4,000 characters, but you can use the "max" keyword to indicate it as essentially a very large character field (up to 2^31 bytes of data).
`Ntext`	Unicode	Varies	Variable-length Unicode character data. Like the `Text` data type, this is legacy support only. In this case, use `nvarchar(max)`.
`Binary`	Binary	Varies	Fixed-length binary data with a maximum length of 8,000 bytes
`VarBinary`	Binary	Varies	Variable-length binary data with a maximum specified length of 8,000 bytes, but you can use the "max" keyword to indicate it as essentially a LOB field (up to 2^31 bytes of data).
`Image`	Binary	Varies	Legacy support only as of SQL Server 2005. Use `varbinary(max)` instead!
`Table`	Other	Special	This is primarily for use in working with result sets, typically passing one out of a User-Defined Function or as a parameter for stored procedures. Not usable as a data type within a table definition (you can't nest tables).
`HierarchyID`	Other	Special	Special data type that maintains hierarchy-positioning information. Provides special functionality specific to hierarchy needs. Comparisons of depth, parent/child relationships, and indexing are allowed. Exact size varies with the number and average depth of nodes in the hierarchy.

Continued

Data type Name	Class	Size in Bytes	Nature of the Data
`Sql_variant`	Other	Special	This is loosely related to the `Variant` in VB and C++. Essentially, it is a container that allows you to hold most other SQL Server data types in it. That means you can use this when one column or function needs to be able to deal with multiple data types. Unlike VB, using this data type forces you to *explicitly* cast it in order to convert it to a more specific data type.
`XML`	Character	Varies	Defines a character field as being for XML data. Provides for the validation of data against an XML Schema as well as the use of special XML-oriented functions.
`CLR`	Other	Varies	Varies depending on the specific nature of the CLR object supporting a CLR based custom data type.

Most of these have equivalent data types in other programming languages. For example, an `int` in SQL Server is equivalent to a `Long` in Visual Basic, and, for most systems and compiler combinations in C++, is equivalent to a signed int.

SQL Server has no concept of unsigned numeric data types.

In general, SQL Server data types work much as you would expect given experience in most other modern programming languages. Adding numbers yields a sum, but adding strings concatenates them. When you mix the usage or assignment of variables or fields of different data types, a number of types convert *implicitly* (or automatically). Most other types can be converted explicitly. (You specifically say what type you want to convert to.) A few can't be converted between at all. Figure 1-3 contains a chart that shows the various possible conversions:

Why would we have to convert a data type? Well, let's try a simple example. If I wanted to output the phrase, `Today's date is ##/##/####`, where ##/##/#### is the current date, I could write it like this:

```
SELECT 'Today''s date is ' + GETDATE()
```

We will discuss Transact-SQL statements such as this in much greater detail later in the book, but the expected result of the previous example should be fairly obvious to you.

The problem is that this statement would yield the following result:

```
Msg 241, Level 16, State 1, Line 1
Conversion failed when converting date and/or time from character string.
```

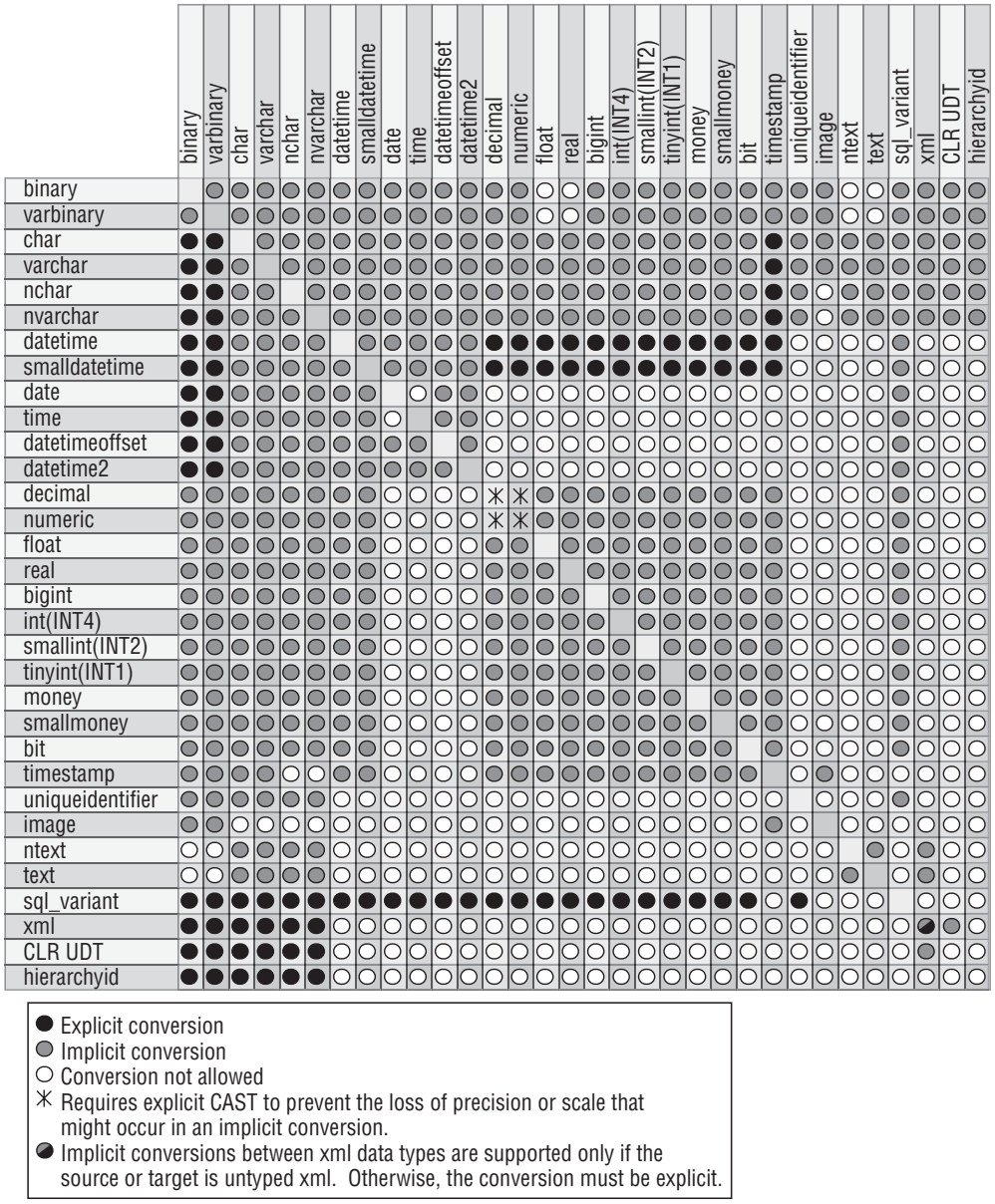

Figure 1-3

Not exactly what we were after, is it? Now let's try it with the `CONVERT()` function:

```
SELECT 'Today''s date is ' + CONVERT(varchar(12), GETDATE(),101)
```

Using `CONVERT` like this yields something like:

```
-----------------------------------
Today's date is 01/01/2008

(1 row(s) affected)
```

Date and time data types, such as the output of the `GETDATE()` function, aren't implicitly convertible to a string data type, such as `Today's date is`, yet we run into these conversions on a regular basis. Fortunately, the `CAST` and `CONVERT()` functions enable us to convert between many SQL Server data types. We will discuss the `CAST` and `CONVERT()` functions more in a later chapter.

In short, data types in SQL Server perform much the same function that they do in other programming environments. They help prevent programming bugs by ensuring that the data supplied is of the same nature that the data is supposed to be (remember 1/1/1980 means something different as a date than as a number) and ensures that the kind of operation performed is what you expect.

NULL Data

What if you have a row that doesn't have any data for a particular column — that is, what if you simply don't know the value? For example, let's say that we have a record that is trying to store the company performance information for a given year. Now, imagine that one of the fields is a percentage growth over the prior year, but you don't have records for the year before the first record in your database. You might be tempted to just enter a zero in the `PercentGrowth` column. Would that provide the right information though? People who didn't know better might think that meant you had zero percent growth, when the fact is that you simply don't know the value for that year.

Values that are indeterminate are said to be `NULL`. It seems that every time I teach a class in programming, at least one student asks me to define the value of `NULL`. Well, that's a tough one, because by definition a `NULL` value means that you don't know what the value is. It could be 1. It could be 347. It could be –294 for all we know. In short, it means *undefined* or perhaps *not applicable*.

SQL Server Identifiers for Objects

Now you've heard all sorts of things about objects in SQL Server. Let's take a closer look at naming objects in SQL Server.

What Gets Named?

Basically, everything has a name in SQL Server. Here's a partial list:

- Stored procedures
- Tables
- Columns
- Views
- Rules
- Constraints
- Defaults
- Indexes
- Filegroups
- Triggers
- Databases
- Servers
- User-defined functions
- Logins
- Roles
- Full-text catalogs
- Files
- User-defined types

And the list goes on. Most things I can think of except rows (which aren't really objects) have a name. The trick is to make every name both useful and practical.

Rules for Naming

As I mentioned earlier in the chapter, the rules for naming in SQL Server are fairly relaxed, allowing things like embedded spaces and even keywords in names. Like most freedoms, however, it's easy to make some bad choices and get yourself into trouble.

Here are the main rules:

- The name of your object must start with any letter, as defined by the specification for Unicode 3.2. This includes the letters most Westerners are used to: A–Z and a–z. Whether "A" is different than "a" depends on the way your server is configured, but either makes for a valid beginning to an object name. After that first letter, you're pretty much free to run wild; almost any character will do.
- The name can be up to 128 characters for normal objects and 116 for temporary objects.
- Any names that are the same as SQL Server keywords or contain embedded spaces must be enclosed in double quotes ("") or square brackets ([]). Which words are considered keywords varies depending on the compatibility level to which you have set your database.

Note that double quotes are only acceptable as a delimiter for column names if you have `SET QUOTED_IDENTIFIER ON`*. Using square brackets (*`[` *and* `]`*) avoids the chance that your users will have the wrong setting.*

These rules are generally referred to as the rules for identifiers and are in force for any objects you name in SQL Server, but may vary slightly if you have a localized version of SQL Server (one adapted for certain languages, dialects, or regions). Additional rules may exist for specific object types.

I'm going to take this as my first opportunity to launch into a diatribe on the naming of objects. SQL Server has the ability to embed spaces in names and, in some cases, to use keywords as names. Resist the temptation to do this! Columns with embedded spaces in their name have nice headers when you make a `SELECT` statement, but there are other ways to achieve the same result. Using embedded spaces and keywords for column names is literally begging for bugs, confusion, and other disasters. I'll discuss later why Microsoft has elected to allow this, but for now, just remember to associate embedded spaces or keywords in names with evil empires, torture, and certain death. (This won't be the last time you hear from me on this one.)

Summary

Like most things in life, the little things do matter when thinking about an RDBMS. Sure, almost anyone who knows enough to even think about picking up this book has an idea of the *concept* of storing data in columns and rows, even if they don't know that these groupings of columns and rows should be called tables. But a few tables seldom make a real database. The things that make today's RDBMSs great are the extra things — the objects that enable you to place functionality and business rules that are associated with the data right into the database with the data.

Database data has *type*, just as most other programming environments do. Most things that you do in SQL Server are going to have at least some consideration of type. Review the types that are available, and think about how these types map to the data types in any programming environment with which you are familiar.

2

Tools of the Trade

Now that we know something about the many types of objects that exist in SQL Server, we probably should get to know something about how to find these objects, and how to monitor your system in general.

In this chapter, we will look into the tools that serve SQL Server's base engine (the relational database engine — tools for managing the add-on services in the chapters where we cover each of those services). Some of them offer only a small number of highly specialized tasks; others do many different things. Most of them have been around in SQL Server in one form or another for a long time.

The tools we will look at in this chapter will be:

- SQL Server Books Online
- SQL Server Configuration Manager
- SQL Server Management Studio
- SQL Server Integration Services (SSIS), including the Import/Export Wizard
- Database Engine Tuning Advisor
- Reporting Services Configuration Manager
- Bulk Copy Program (bcp)
- Profiler
- sqlcmd
- PowerShell

Books Online

Is *Books Online* a tool? I think so. Let's face it. It doesn't matter how many times you read this or any other book on SQL Server; you're not going to remember everything you'll ever need to know about SQL Server. SQL Server is one of my mainstay products, and I still can't remember it all. Books Online is simply one of the most important tools you're going to find in SQL Server.

My general philosophy about books or any other reference materials related to programming is that I can't have enough of them. I first began programming in 1980 or so, and back then it was possible to remember most things (but not everything). Today it's simply impossible. If you have any diversification at all (something that is, in itself, rather difficult these days), there are just too many things to remember, and the things you don't use every day get lost in dying brain cells.

Here's a simple piece of advice: Don't even try to remember it all. Remember what you've seen is possible. Remember what is an integral foundation to what you're doing. Remember what you work with every day. Then remember to build a good reference library (starting with this book) and keep a healthy list of good SQL Server sites in your favorites list to fill in information on subjects you don't work with every day and may not remember the details of.

As you see in Figure 2-1, Books Online in SQL Server uses the updated .NET online help interface, which is replacing the older standard online help interface used among the Microsoft technical product line (Back Office, MSDN, Visual Studio).

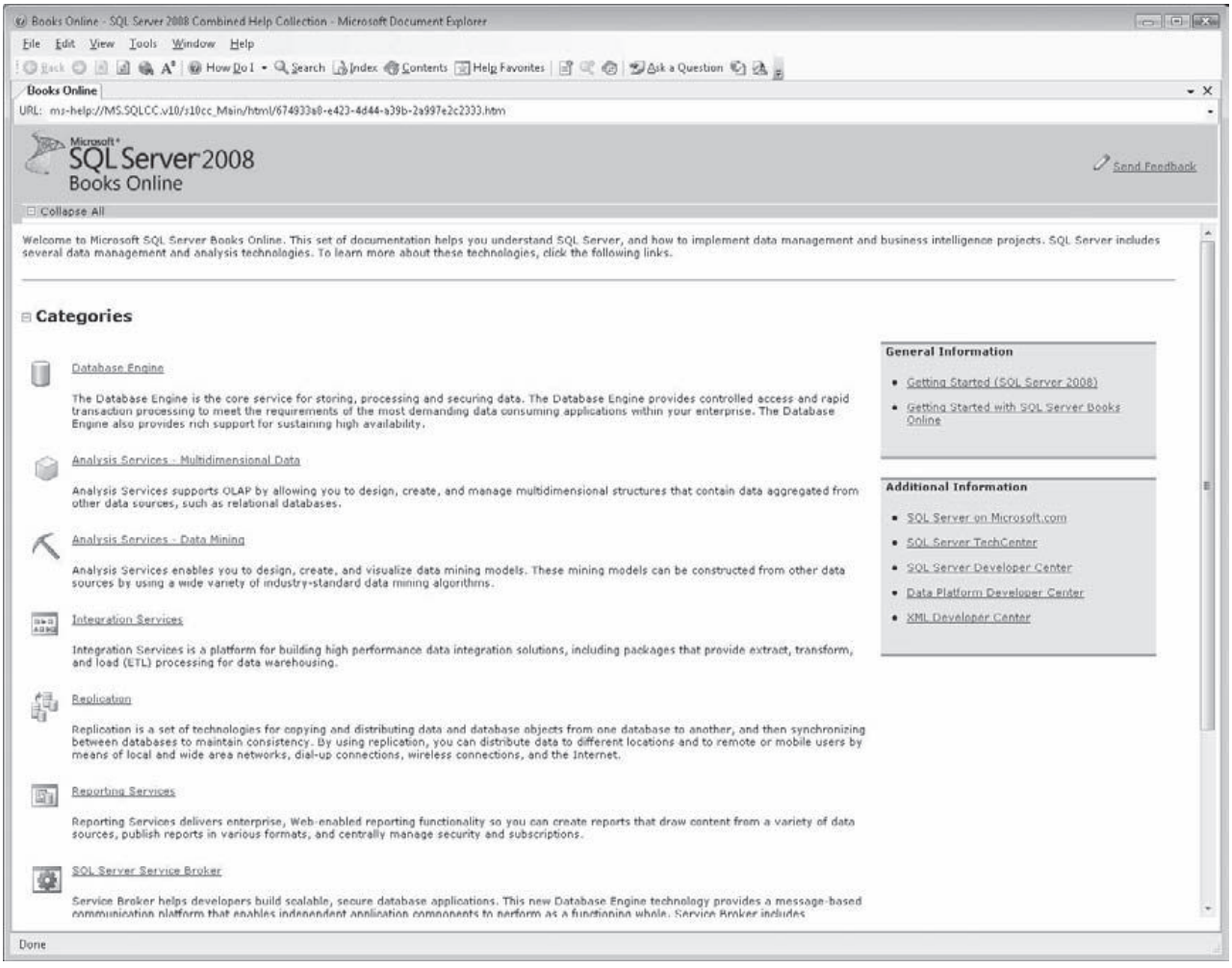

Figure 2-1

Everything works pretty much as one would expect here, so I'm not going to go into the details of how to operate a help system. Suffice it to say that SQL Server Books Online is a great quick reference that follows you to whatever machine you're working on at the time.

> **Technically speaking, it's quite possible that not every system you move to will have Books Online (BOL) installed. This is because you can manually deselect BOL at the time of installation. Even in tight space situations, however, I strongly recommend that you always install BOL. It really doesn't take up all that much space when you consider cost per megabyte these days, and having that quick reference available wherever you are running SQL Server can save you a fortune in time. (On my machine, Books Online takes up roughly 110MB of space.)**

SQL Server Configuration Manager

Administrators who configure computers for database access are the main users of this tool, but it's still important to understand what this tool is about.

The SQL Server Configuration Manager combines some settings that were, in earlier releases, spread across multiple tools into one spot. The items managed in the Configuration Manager fall into two areas:

- Service Management
- Network Configuration

Note that, while I'm only showing two major areas here, the Configuration Manager splits the Network Configuration side of things up into multiple nodes.

Service Management

SQL Server is a large product and the various pieces of it utilize a host of services that run in the background on your server. A full installation will encompass nine different services, and seven of these can be managed from this part of the SQL Server Configuration Manager (the other two are services that act as background support).

The services available for management here include:

- **Integration Services** — This powers the Integration Services engine that you look at in Chapter 18.
- **Analysis Services** — This powers the Analysis Services engine.
- **Reporting Services** — The underlying engine that supports Reporting Services.
- **SQL Server Agent** — The main engine behind anything in SQL Server that is scheduled. Utilizing this service, you can schedule jobs to run on a variety of different schedules. These jobs can have multiple tasks to them and can even branch into different tasks depending on the outcome of some previous task. Examples of things run by the SQL Server Agent include backups, as well as routine import and export tasks.

- **SQL Server** — The core database engine that works on data storage, queries, and system configuration for SQL Server.
- **SQL Server Browser** — This supports advertising your server so those browsing your local network can identify that your system has SQL Server installed.

Network Configuration

A fair percentage of the time, any connectivity issues discovered are the result of client network configuration, or how that configuration matches with that of the server.

SQL Server provides several of what are referred to as *Net-Libraries* (network libraries), or *NetLibs*. These are dynamic-link libraries (DLLs) that SQL Server uses to communicate with certain network protocols. NetLibs serve as something of an insulator between your client application and the network protocol, which is essentially the language that one network card uses to talk to another, that is to be used. They serve the same function at the server end, too. The NetLibs supplied with SQL Server 2008 include:

- Named Pipes
- TCP/IP (the default)
- Shared Memory
- VIA (a special virtual interface that your storage-hardware vendor may support)

VIA is a special network library that is made for use with some very special (and expensive) hardware. If you're running in a VIA environment, you'll know about the special requirements associated with it. For those of you that aren't running in that environment, it suffices to say that VIA offers a very fast but expensive solution to high-speed communication between servers. It would not usually be used for a normal client.

The same NetLib must be available on both the client and server computers so that they can communicate with each other via the network protocol. Choosing a client NetLib that is not also supported on the server will result in your connection attempt failing with a `Specified SQL Server Not Found` error.

Regardless of the data access method and kind of driver used (SQL Native Client, ODBC, OLE DB), it will always be the driver that talks to the NetLib. The process works as shown in Figure 2-2. The steps in order are:

1. The client app talks to the driver (SQL Native Client, ODBC).
2. The driver calls the client NetLib.
3. This NetLib calls the appropriate network protocol and transmits the data to a server NetLib.
4. The server NetLib then passes the requests from the client to SQL Server.

In case you're familiar with TCP/IP, the default port that the IP NetLib will listen on is 1433. A port can be thought of as being like a channel on the radio — signals are bouncing around on all sorts of different frequencies, but they only do you any good if you're *listening* on the right channel. Note that this is the default, so there is no guarantee that the particular server you're trying to connect to is listening to that particular port — indeed, most security experts recommend changing it to something nonstandard.

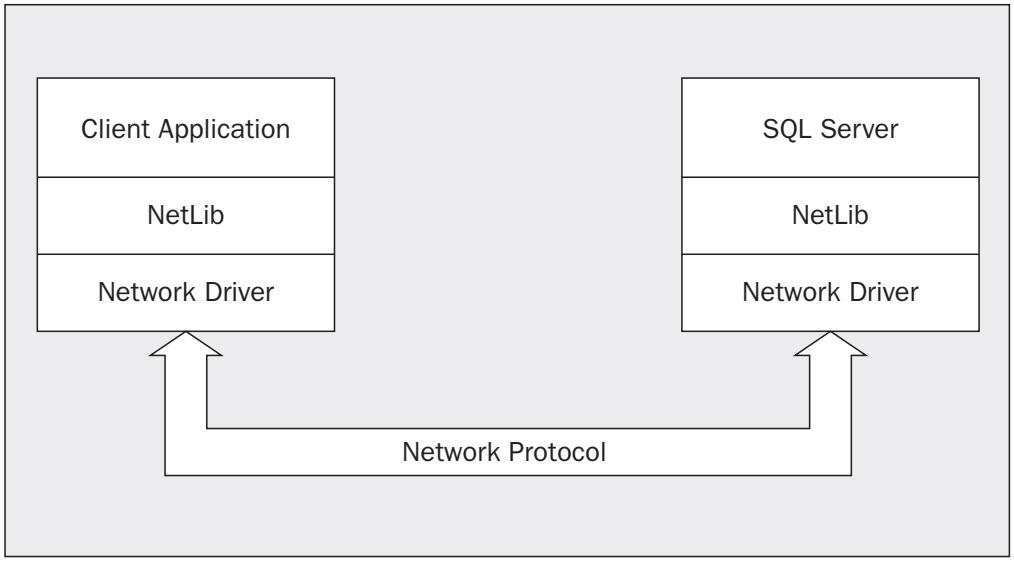

Figure 2-2

Replies from SQL Server to the client follow the same sequence, only in reverse.

The Protocols

Let's start off with that "What are the available choices?" question. If you run the Configuration Management utility and open the Server Network Configuration tree, you'll see something like Figure 2-3.

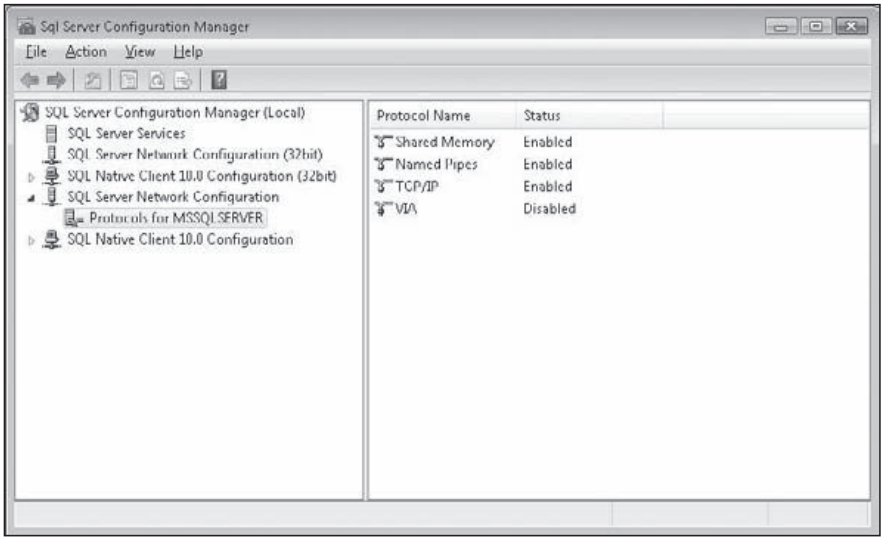

Figure 2-3

How many nodes are shown here will vary depending on your installation. In Figure 2-3, there are duplicate nodes for Network and Client configuration to allow for separate treatment of 32bit vs. 64bit libraries. There will only be one node for each if you're running a 32bit installation.

For security reasons, only Shared Memory is enabled at installation time.

You'll want to leave Shared Memory enabled for when you're accessing the machine locally. (It works only when the client is on the same physical server as the SQL Server installation.) But you need to enable at least one other NetLib if you want to be able to contact your SQL Server remotely (say, from a Web server or from different clients on your network).

Keep in mind that, in order for your client to gain a connection to the server, the server has to be listening for the protocol with which the client is trying to communicate and, in the case of TCP/IP, on the same port.

At this point, you might be tempted to say, "Hey, why don't I just enable every NetLib? Then I won't have to worry about it." This situation is like anything you add onto your server — more overhead. In this case, it would both slow down your server (not terribly, but every little bit counts) and expose you to unnecessary openings in your security. (Why leave an extra door open if nobody is supposed to be using that door?)

OK, now let's take a look at what we can support and why we would want to choose a particular protocol.

Named Pipes

Named Pipes can be very useful when TCP/IP is not available, or there is no Domain Name Service (DNS) server to allow the naming of servers under TCP/IP.

Technically speaking, you can connect to a SQL Server running TCP/IP by using its IP address in the place of the name. This works all the time, even if there is no DNS service, as long as you have a route from the client to the server. (If it has the IP address, then it doesn't need the name.) Keep in mind, however, that if your IP address changes for some reason, you'll need to change what IP address you're accessing (a real pain if you have a bunch of config files you need to go change!).

TCP/IP

TCP/IP has become something of the de facto standard networking protocol and is also the only option if you want to connect directly to your SQL Server via the Internet, which, of course, uses only IP.

Don't confuse the need to have your database server available to a Web server with the need to have your database server directly accessible to the Internet. You can have a Web server that is exposed to the Internet, but also has access to a database server that is not directly exposed to the Internet. (The only way for an Internet connection to see the data server is through the Web server.)

Connecting your data server directly to the Internet is a security hazard in a big way. If you insist on doing it (and there can be valid reasons for doing so, rare though they may be), then pay particular attention to security precautions.

Shared Memory

Shared memory removes the need for inter-process marshaling — a way of packaging information before transferring it across process boundaries — between the client and the server, if they are running on the same box. The client has direct access to the same memory-mapped file where the server is storing data. This removes a substantial amount of overhead and is *very* fast. It's only useful when accessing the server locally (say, from a Web server installed on the same server as the database), but it can be quite a boon performance-wise.

On to the Client

Now we've seen all the possible protocols and we know how to choose which ones to offer. Once we know what our server is offering, we can go and configure the client. Most of the time, the defaults are going to work just fine, but let's take a look at what we've got. Expand the Client Network Configuration tree and select the Client Protocols node, as shown in Figure 2-4.

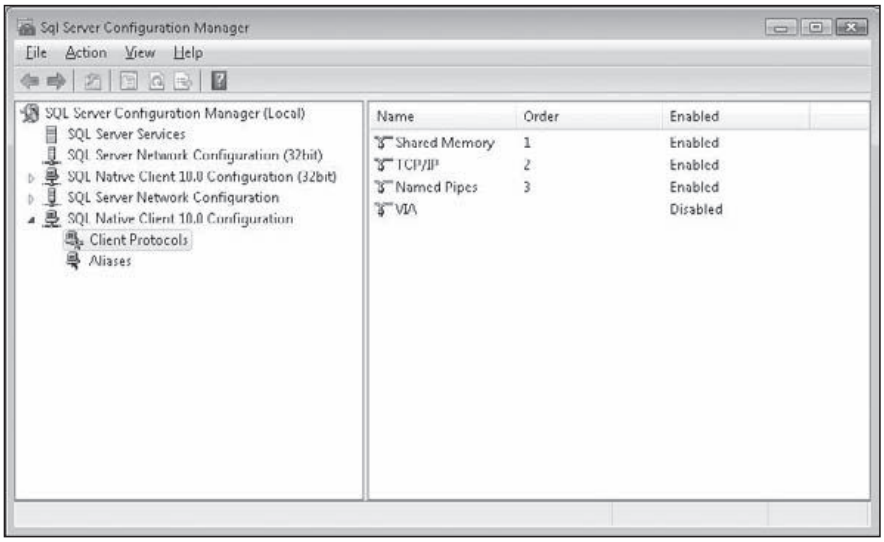

Figure 2-4

SQL Server has the ability for the client to start with one protocol, then, if that doesn't work, move on to another. In Figure 2-4, I am first using Shared Memory, then trying TCP/IP, and finally going to Named Pipes if TCP/IP doesn't work as defined by the Order column. Unless you change the default (changing the priority by using the up and down arrows), Shared Memory is the NetLib that is used first for connections to any server not listed in the aliases list (the next node under Client Network Configuration), followed by TCP/IP and so on.

If you have TCP/IP support on your network, configure your server to use it for any remote access. IP has less overhead and just plain runs faster; there is no reason not to use it, unless your network doesn't support it. It's worth noting, however, that for local servers (where the server is on the same physical system as the client), the Shared Memory NetLib will be quicker, as you do not need to go through the network stack to view your local SQL Server.

The Aliases list is a listing of all the servers on which you have defined a specific NetLib to be used when contacting that particular server. This means that you can contact one server using IP and another using Named Pipes — whatever you need to get to that particular server. Figure 2-5 shows a client configured to use the Named Pipes NetLib for requests from the server named `HOBBES` and to use whatever is set up as the default for contact with any other SQL Server.

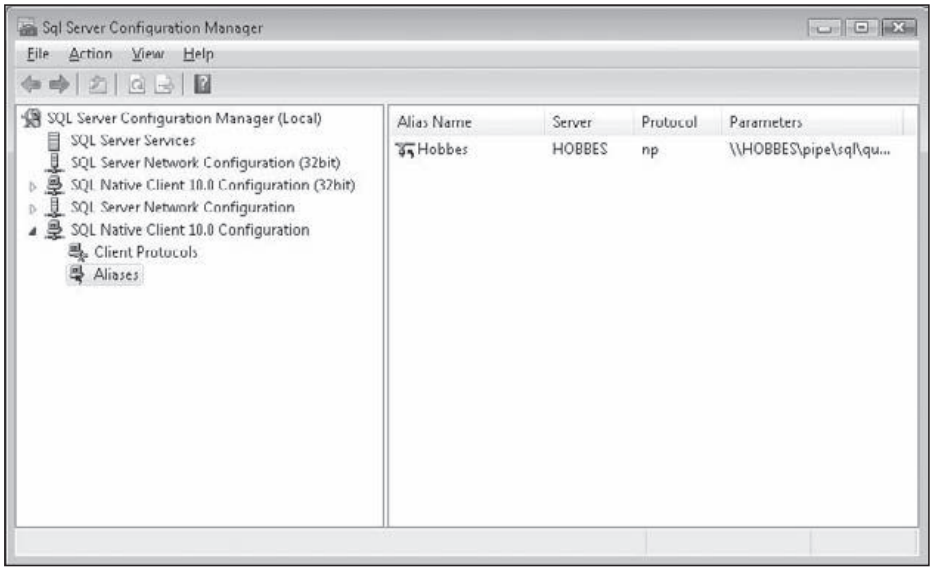

Figure 2-5

Again, remember that the Client Network Configuration setting on the network machine must have a default protocol that matches one supported by the server, or it must have an entry in the Aliases list to specifically choose a NetLib supported by that server.

> *If you are connecting to your SQL Server over the Internet (which is a very bad idea from a security standpoint, but people do it), you'll probably want to use the server's actual IP address, rather than the name of the server. This gets around some name resolution issues that may occur when dealing with SQL Server and the Internet. Keep in mind, however, that you'll need to change the IP address manually if the server gets a new IP; you won't be able to count on DNS to take care of it for you.*

SQL Server Management Studio

The *SQL Server Management Studio* is pretty much home base when administering a SQL Server. It provides a variety of functionality for managing your server using a relatively easy-to-use graphical user interface. Branched off of the Visual Studio IDE environment's code base, it combines a myriad of functionality that used to be in separate tools.

For the purposes of this book, we're not going to cover everything that the Management Studio has to offer, but let's make a quick run down of the things you can do:

- Create, edit, and delete databases and database objects
- Manage scheduled tasks, such as backups and the execution of SSIS package runs

- Display current activity, such as who is logged on, what objects are locked, and from which client they are running
- Manage security, including such items as roles, logins, and remote and linked servers
- Initiate and manage the Database Mail Service
- Create and manage full-text search catalogs
- Manage configuration settings for the server
- Initiate an instance of the new PowerShell console
- Create and manage publishing and subscribing databases for replication

We will be seeing a great deal of the Management Studio throughout this book, so let's take a closer look at some of the key functions Management Studio serves.

Getting Started with the Management Studio

When you first start the Management Studio, you are presented with a Connection dialog box similar to the one in Figure 2-6.

Figure 2-6

Your login screen may look a little bit different from this, depending on whether you've logged in before, what machine you logged into, and what login name you used. Most of the options on the login screen are pretty self-explanatory, but let's look at a couple in more depth.

Server Type

This relates to which of the various subsystems of SQL Server you are logging in to (the normal database server, Analysis Services, Report Server, or Integration Services). Since these different types of servers can share the same name, pay attention to this to make sure you're logging in to what you think you're logging in to.

Server Name

As you might guess, this is the SQL Server in to which you're asking to be logged. In Figure 2-6, we have chosen ".". This doesn't mean that there is a server named period, but rather that we want to

log in to the default instance of SQL Server that is on this same machine, regardless of what this machine is named. Selecting "." (local) not only automatically identifies which server (and instance) you want to use, but also how you're going to get there. You can also use "(local)" as another option that has the same meaning as ".".

> **SQL Server allows multiple *instances* of SQL Server to run at one time. These are just separate loads into memory of the SQL Server engine running independently from each other.**

Note that the default instance of your server will have the same name as your machine on the network. There are ways to change the server name after the time of installation, but they are problematic at best, and deadly to your server at worst. Additional instances of SQL Server will be named the same as the default (HOBBES or KIERKEGAARD in many of the examples in this book) followed by a dollar sign, and the instance name, for example, SIDDARTHA$SHRAMANA.

If you select "." or (local), your system uses the Shared Memory NetLib regardless of which NetLib you selected for contacting other servers. This is a bad news/good news story. The bad news is that you give up a little bit of control. (SQL Server will always use Shared Memory to connect; you can't choose anything else.) The good news is that you don't have to remember which server you're on and you get a high-performance option for work on the same machine. If you use your local PC's actual server name, your communications will still go through the network stack and incur the overhead associated with that, just as if you were communicating with another system, regardless of the fact that it is on the same machine.

Now what if you can't remember what the server's name is? Just click the down arrow to the right of the server box to get a list of recently connected servers. If you scroll down, you'll see a Browse for More option. If you choose this option, SQL Server will poll the network for any servers that are advertising to the network; essentially, this is a way for a server to let itself be known to other systems on the network. You can see from Figure 2-7 that you get two tabs: one that displays local servers (all of the instances of SQL Server on the same system you're running on) and another that shows other SQL Servers on the network.

You can select one of these servers and click OK.

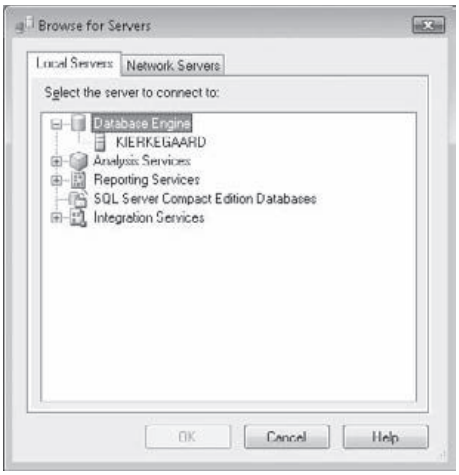

Figure 2-7

Watch out when using the Server selection dialog box. Although it's usually pretty reliable, there are ways of configuring a SQL Server so that it doesn't broadcast. When a server has been configured this way, it won't show up in the list. Also, servers that are only listening on the TCP/IP NetLib and don't have a DNS entry will not show up. You must, in this case, already know your IP address and refer to the server using it.

Authentication

You can choose between Windows Authentication and SQL Server Authentication. Windows Authentication will always be available, even if you configured it as SQL Server Authentication. Logins using usernames and passwords that are local to SQL Server (not part of a larger Windows network) are acceptable to the system only if you specifically turn on SQL Server Authentication.

Windows Authentication

Windows Authentication is just as it sounds. You have Windows users and groups. Those Windows users are mapped into SQL Server logins in their Windows user profile. When they attempt to log in to SQL Server, they are validated through the Windows domain and mapped to roles according to the login. These roles identify what the user is allowed to do.

The best part of this model is that you have only one password. (If you change it in the Windows domain, then it's changed for your SQL Server logins, too.) You don't have to fill in anything to log in; it just takes the login information from the way you're currently logged in to the Windows network. Additionally, the administrator has to administer users in only one place. The downside is that mapping this process can get complex and, to administer the Windows user side of things, you must be a domain administrator.

SQL Server Authentication

The security does not care at all about what the user's rights to the network are, but rather what you explicitly set up in SQL Server. The authentication process doesn't take into account the current network login at all; instead, the user provides a SQL Server–specific login and password.

This can be nice because the administrator for a given SQL Server doesn't need to be a domain administrator (or even have a username on your network, for that matter) to give rights to users on the SQL Server. The process also tends to be somewhat simpler than under Windows Authentication. Finally, it means that one user can have multiple logins that give different rights to different things.

Try It Out — Making the Connection

Let's get logged in.

1. Choose the (local) option for the SQL Server.
2. Select SQL Server Authentication.

3. Select a login name of sa, which stands for System Administrator. Alternatively, you may log in as a different user, as long as that user has system administrator privileges.

4. Enter the same password that was set when you installed SQL Server. On case-sensitive servers, the login is also case sensitive, so make sure you enter it in lowercase.

If you're connecting to a server that has been installed by someone else, or where you have changed the default information, you need to provide login information that matches those changes. After you click OK, you should see the Object Explorer window screen shown in Figure 2-8.

> **Be careful with the password for the sa user. This and any other user who is a sysadmin is a super-user with full access to everything.**

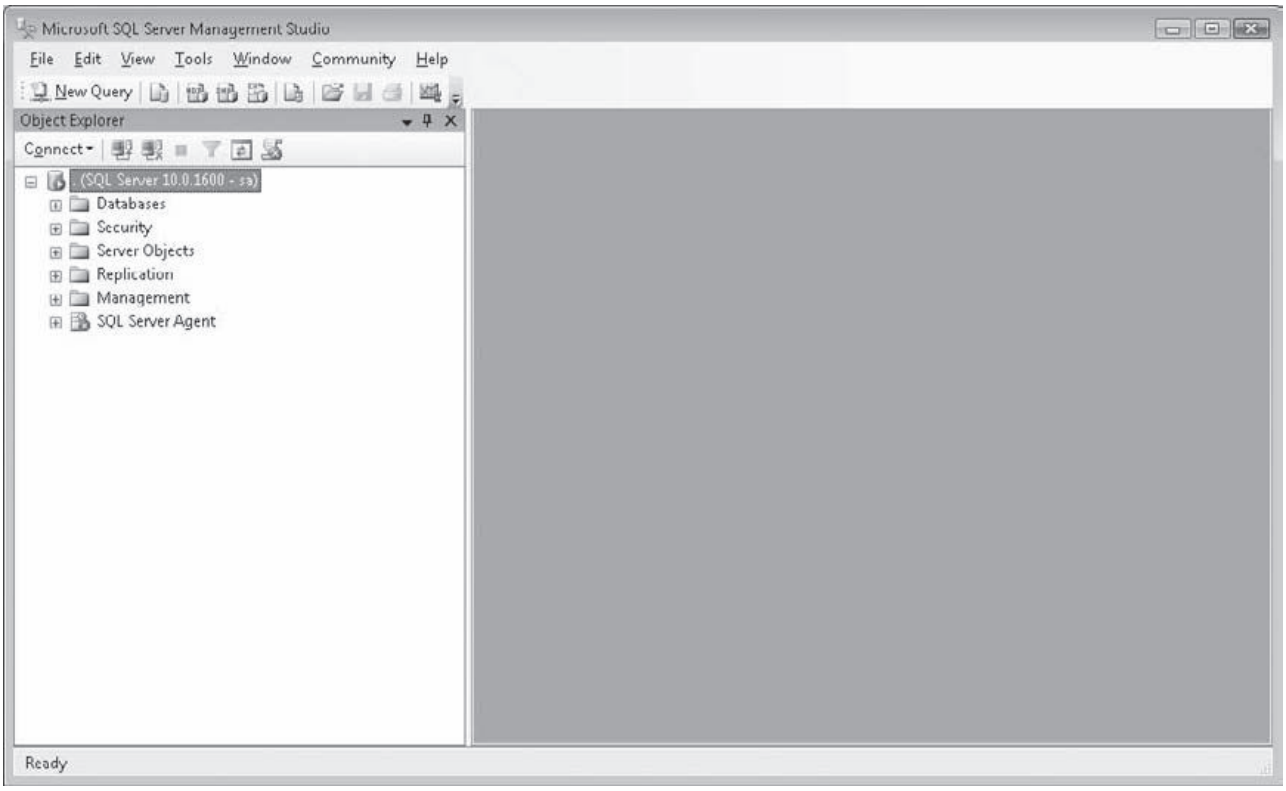

Figure 2-8

How It Works

The Login dialog gathers all the information needed to create the connection. Once it has that, it assembles the connection information into a single connection string and sends that to the server. The connection is then either accepted or rejected and, if it is accepted, a connection handle is given to the Query window so that the connection information can be used over and over again for many queries, as long as you do not disconnect.

Again, many of the items here (New, Open, Save, Cut, Paste, and so on) are things that you have seen plenty of times in other Windows applications and should be familiar with, but there's also a fair amount that's specific to SQL Server. The main thing to notice for now is that the menus in the Management Studio are context sensitive — that is, different menus are available and what they contain changes based on which window is active in the Studio. Be sure to explore the different contextual menus you get as you explore different parts of the Management Studio.

Query Window

This part of the Management Studio takes the place of what was, at one time, a separate tool that was called *Query Analyzer*. The Query window is your tool for interactive sessions with a given SQL Server. It's where you can execute statements using *Transact-SQL* (T-SQL). I lovingly pronounce it "Tee-Squeal," but it's supposed to be "Tee-Sequel." T-SQL is the native language of SQL Server. It's a dialect of Structured Query Language (SQL), and is largely compliant with modern ANSI/ISO SQL standards. You'll find that most RDBMS products support basic ANSI/ISO SQL compatibility.

Because the Query window is where we will spend a fair amount of time in this book, let's take a more in-depth look at this tool and get familiar with how to use it.

Getting Started

Well, I've been doing plenty of talking about things in this book, and it's high time we started doing something. To that end, open a new Query window by clicking the New Query button towards the top-left of the Management Studio, or choosing File ➪ New ➪ New Query With Current Connection from the File menu. When the Query window opens, you'll get menus that largely match those in Query Analyzer back when that was a separate tool. We will look at the specifics, but let's get our very first query out of the way.

Start by selecting `AdventureWorks2008` in the database drop-down box on the SQL Editor toolbar, then type the following code into the main window of the Query window:

```
SELECT * FROM Person.Address;
```

Notice several things happen as you type:

- The coloring of words and phrases changes as you type.
- As you type, the Management Studio guesses at what you're trying to do (as shown in Figure 2-9). Utilizing *IntelliSense*, much like Visual Studio, SQL Server will give you hints as to what probably should come next in your code.

Statement keywords should appear in blue. Unidentifiable items, such as column and table names (these vary with every table in every database on every server), are in black. Statement arguments and connectors are in red. Pay attention to how these work and learn them. They can help you catch many bugs before you've even run the statement (and seen the resulting error).

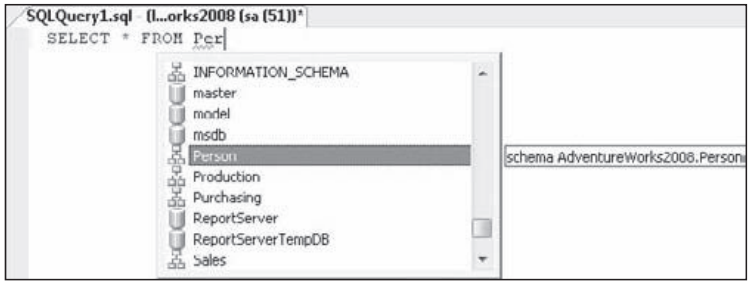

Figure 2-9

Note that IntelliSense is new with SQL Server 2008. While they have done a terrific job with it, it is not without some peculiarities created by the nature of SQL versus other languages. Of particular importance is what help you can get when you're adding columns to be selected. We'll see more about the syntax of this in later chapters, but SQL syntax calls for column names before the names of the tables those columns are sourced from. The result is problematic for IntelliSense as, when you are typing your column names, the tool has no way of knowing what tables you're trying to get those columns from (and therefore no way of giving you appropriate hints). If you're desperate, you can get around this by skipping ahead to add the table names, then coming back to fill in the column names.

The check-mark icon (Parse) on the SQL Editor toolbar represents another simple debugging item that quickly parses the query for you without actually attempting to run the statement. If there are any syntax errors, this should catch them before you see error messages. A debugger is available as another way to find errors. We'll look at that in depth in Chapter 12.

Now click the Execute button (with the red exclamation point next to it) on the toolbar. The Query window changes a bit, as shown in Figure 2-10.

Notice that the main window has been automatically divided into two panes. The top is your original query text; the bottom is called the *results pane*. In addition, notice that the results pane has a tab at the top of it. Later on, after we've run queries that return multiple sets of data, you'll see that we can get each of these results on separate tabs; this can be rather handy, because you often don't know how long each set of data, or *result set*, is.

The terms result set and recordset are frequently used to refer to a set of data that is returned as a result of some command being run. You can think of these words as interchangeable.

Now change a setting or two and see how what we get varies. Take a look at the toolbar above the Query window and check out a set of three icons, highlighted in Figure 2-11.

These control the way you receive output. In order, they are Results to Text, Results to Grid, and Results to File. The same choices can also be made from the Query menu under the Results To submenu.

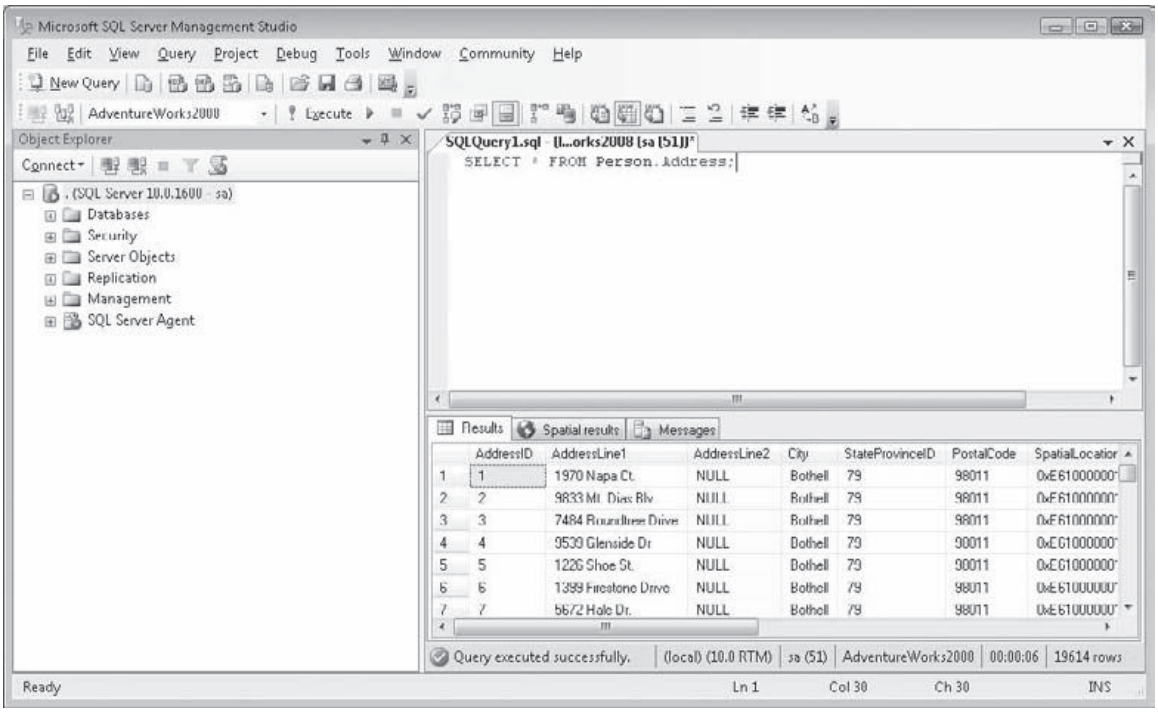

Figure 2-10

Figure 2-11

Results to Text

The Results to Text option takes all the output from your query and puts it into one page of text results. The page can be of virtually infinite length (limited only by the available memory in your system).

Before discussing this further, rerun that previous query using this option and see what you get. Choose the Results to Text option and rerun the previous query by clicking Execute, as shown in Figure 2-12.

The data that you get back is exactly the same as before. It's just given to you in a different format. I use this output method in several scenarios:

- When I'm only getting one result set and the results have only fairly narrow columns
- When I want to be able to save my results in a single text file
- When I'm going to have multiple result sets, but the results are expected to be small, and I want to be able to see more than one result set on the same page without dealing with multiple scrollbars

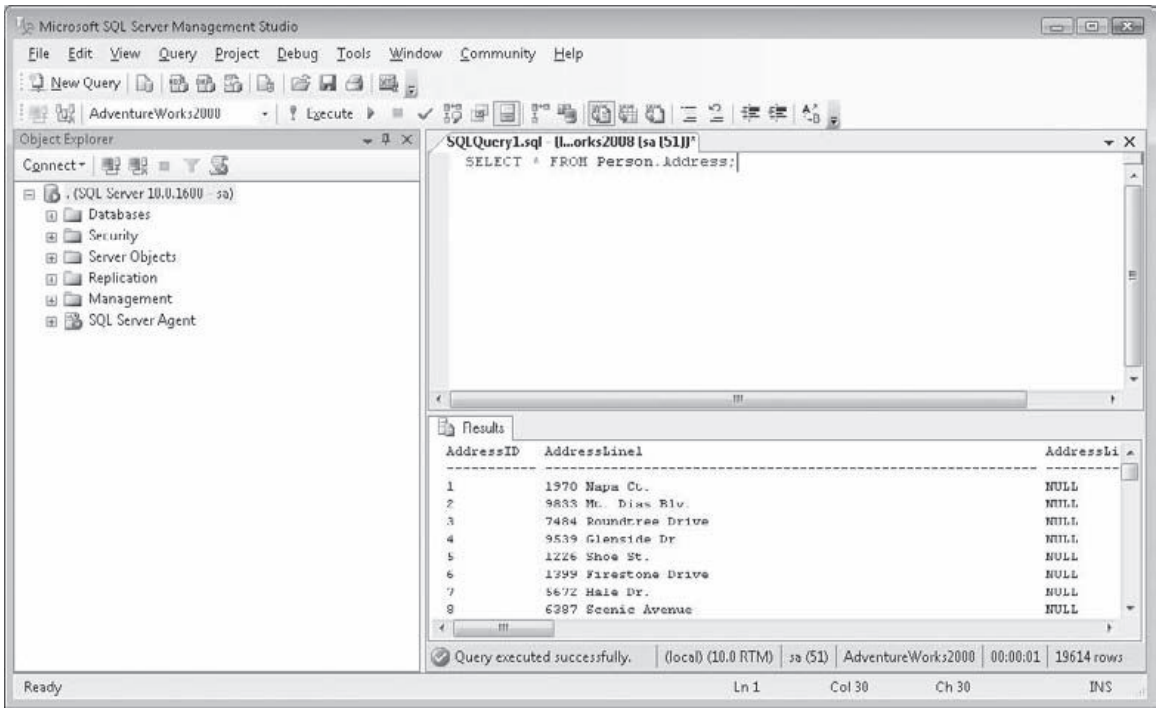

Figure 2-12

Results to Grid

This option divides the columns and rows into a grid arrangement. Following is a list of specific things that this option gives you that the Results to Text doesn't:

- You can resize the column by hovering your mouse pointer on the right border of the column header, then clicking and dragging the column border to its new size. Double-clicking the right border results in the autofit for the column.
- If you select several cells, then cut and paste them into another grid (say, Microsoft Excel), they will be treated as individual cells. (Under the Results to Text option, the cut data is pasted all into one cell.)
- You can select just one or two columns of multiple rows. (Under Results to Text, if you select several rows, all the inner rows have every column selected; you can select only in the middle of the row for the first and last row selected.)

I use this option for almost everything because I find that I usually want one of the benefits I just listed.

Results to File

Think of this one as largely the same as Results to Text, but instead of to screen, it routes the output directly to a file. I use this one to generate files I intend to parse using some utility or that I want to easily e-mail.

sqlcmd Mode

We will discuss sqlcmd a bit more shortly. For now, suffice it to say that it is a tool that helps us run queries from a Windows command line. It has some special scripting abilities that are meant specifically for

command-line scripting. By default, these special script commands are not available in the Query window. Turning sqlcmd mode on activates the special sqlcmd scripting options even in the Query window.

> **Be aware that the Query window always utilizes the `SQLNativeClient` connection method (even when operating in sqlcmd mode), whereas the actual sqlcmd utility will use an OLE DB connection. The result is that you may see slight differences in behavior between running a script using sqlcmd versus using sqlcmd from the Query window. These tend to be corner case differences, and are rarely seen and generally innocuous.**

Show Execution Plan

Every time you run a query, SQL Server parses your query into its component parts and then sends it to the *query optimizer*. The query optimizer is the part of SQL Server that figures out the best way to run your query to balance fast results with minimum impact to other users. When you use the Show Estimated Execution Plan option, you receive a graphical representation and additional information about how SQL Server plans to run your query. Similarly, you can turn on the Include Actual Execution Plan option. Most of the time, this will be the same as the estimated execution plan, but you will occasionally see differences here due to changes that the optimizer decides to make while running the query, as well as changes in the actual cost of running the query versus what the optimizer *thinks* is going to happen.

Let's see what a query plan looks like in our simple query. Click the Include Actual Execution Plan option, and execute the query again, as shown in Figure 2-13.

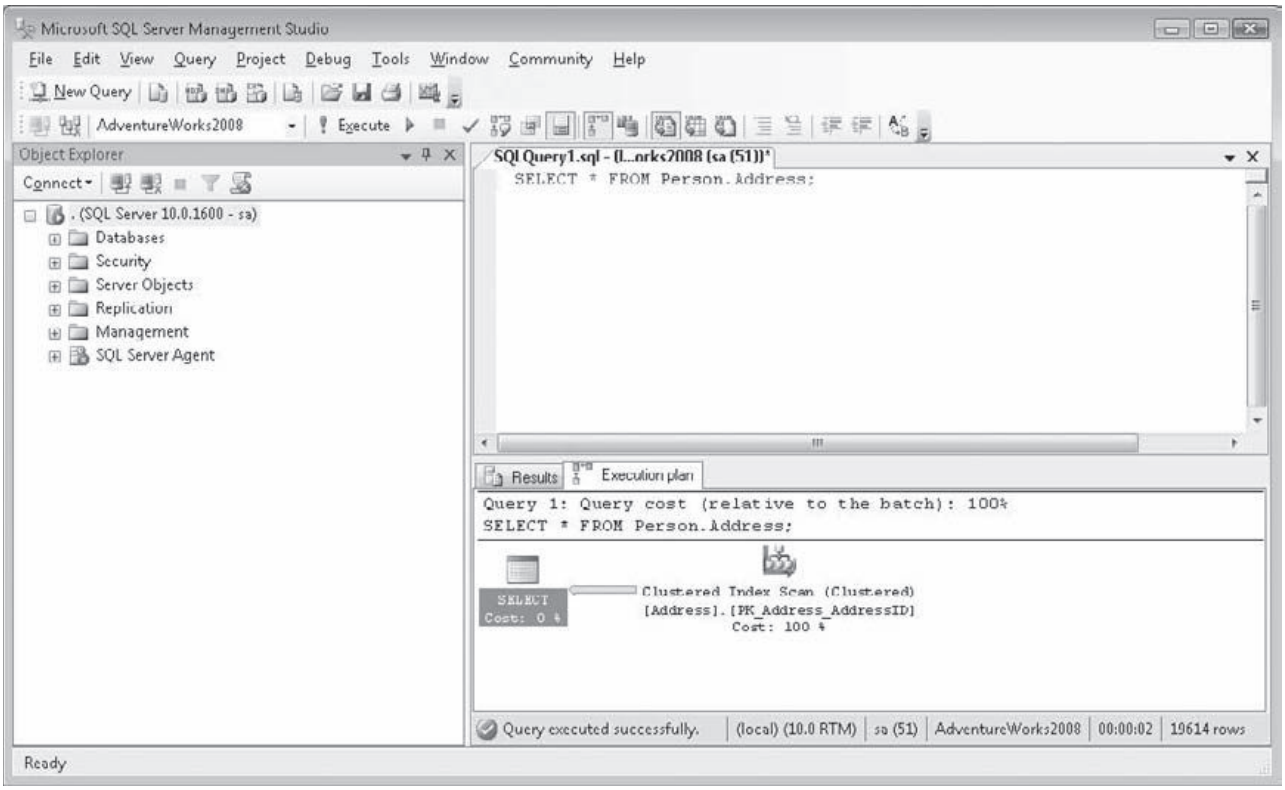

Figure 2-13

Note that you have to actually click the Execution Plan tab for it to come up and that your query results are still displayed in the way you selected. The Show Estimated Execution plan option gives you the same output as an Include Actual Execution Plan does with two exceptions:

- You get the plan immediately rather than after your query executes.
- Although what you see is the actual *plan* for the query, all the cost information is estimated and the query is not actually run. Under Include Actual Execution, the query is physically executed and the cost information you get is actual rather than estimated.

Note that the plan in Figure 2-13 is an extremely simple execution plan. More complex queries may show a variety of branching and parallel operations.

The Available Databases Combo Box

Finally, take another look at the Available Databases combo box. In short, this is where you select the default database that you want your queries to run against for the current window (we changed AdventureWorks2008 to be our default database earlier). Initially, the Query window will start with whatever the default database is for the user that's logged in (for sa, that is the master database unless someone has changed it on your system). You can then change it to any other database that the current login has permission to access. Since we're using the sa user ID, every database on the current server should have an entry in the Available Databases combo box.

The Object Explorer

This useful little tool enables you to navigate your database, look up object names, and even perform actions like scripting and looking at the underlying data.

In the example in Figure 2-14, I've expanded the database node all the way down to the listing of tables in the AdventureWorks2008 database. You can drill down even farther to see individual columns (including data type and similar properties) of the tables — a very handy tool for browsing your database.

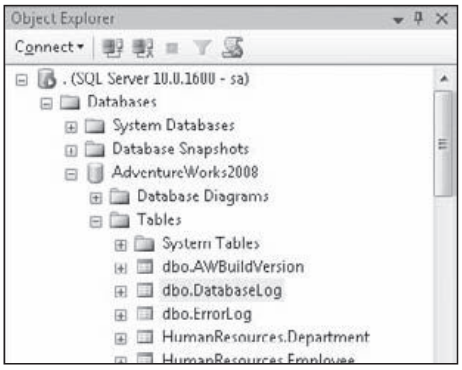

Figure 2-14

SQL Server Integration Services (SSIS)

Your friend and mine — that's what *SSIS* (formerly known as Data Transformation Services or DTS) is. I simply sit back in amazement every time I look at this feature of SQL Server. To give you a touch of perspective here, I've done a couple of Decision Support Systems (DSS) projects over the years. (These are usually systems that don't have online data going in and out, but instead pull data together to help management make decisions.) A DSS project gathers data from a variety of sources and pumps it into one centralized database to be used for centralized reporting.

These projects can get very expensive very quickly, as they attempt to deal with the fact that not every system calls what is essentially the same data by the same name. There can be an infinite number of issues to be dealt with. These can include data integrity (what if the field has a `NULL` and we don't allow `NULL`s?) or differences in business rules (one system deals with credits by allowing a negative order quantity, another doesn't allow this and has a separate set of tables to deal with credits). The list can go on and on, and so can the expense.

With SSIS a tremendous amount of the coding, usually in some client-side language, that had to be done to handle these situations can be eliminated or, at least, simplified. SSIS enables you to take data from any data source that has an OLE DB or .NET data provider and pump it into a SQL Server table.

> **Be aware that there is a special OLE DB provider for ODBC. This provider allows you to map your OLE DB access directly to an ODBC driver. That means anything that ODBC can access can also be accessed by OLE DB (and, therefore, SSIS).**
>
> **While we're at it, it's also worth pointing out that SSIS, although part of SQL Server, can work against any OLE DB source and any OLE DB destination. That means that SQL Server doesn't need to be involved in the process at all other than to provide the data pump. You could, for example, push data from Oracle to Excel, or even DB/2 to MySQL.**

While transferring your data, we can also apply what are referred to as transformations to that data. *Transformations* essentially alter the data according to some logical rule(s). The alteration can be as simple as changing a column name, or as complex as an analysis of the integrity of the data and application of rules to change it if necessary. To think about how this is applied, consider the example I gave earlier of taking data from a field that allows `NULL`s and moving it to a table that doesn't allow `NULL`s. With SSIS you can automatically change any `NULL` values to some other value you choose during the transfer process. (For a number, that might be zero or, for a character, it might be something like unknown.)

Bulk Copy Program (bcp)

If SSIS is your friend and mine, then the *Bulk Copy Program*, or *bcp*, would be that old friend that we may not see that much anymore, but really appreciate when we do.

bcp is a command-line program whose sole purpose in life is to move formatted data in and out of SQL Server en masse. It was around long before what has now become SSIS was thought of, and while SSIS is replacing bcp for most import/export activity, bcp still has a certain appeal for people who like command-line utilities. In addition, you'll find an awful lot of SQL Server installations out there that still depend on bcp to move data around fast.

SQL Server Profiler

I can't tell you how many times this one has saved my bacon by telling me what was going on with my server when nothing else would. It's not something a developer (or even a DBA, for that matter) tends to use every day, but it's extremely powerful and can be your salvation when you're sure nothing can save you.

SQL Server Profiler is, in short, a real-time tracing tool. Whereas Performance Monitor is all about tracking what's happening at the macro level — system configuration stuff — the Profiler is concerned with tracking specifics. This is both a blessing and a curse. The Profiler can, depending on how you configure your trace, give you the specific syntax of every statement executed on your server. Now imagine that you are doing performance tuning on a system with 1000 users. I'm sure you can imagine the reams of paper that would be used to print the statements executed by so many people in just a minute or two. Fortunately, the Profiler has a vast array of filters to help you narrow things down and track more specific problems, such as long-running queries, or the exact syntax of a query being run within a stored procedure. This is nice when your procedure has conditional statements that cause it to run different things under different circumstances.

sqlcmd

As I mentioned back when we were talking about the Management Console, SQL Server has a tool to use when you want to include SQL commands and management tasks in command-line batch files — sqlcmd. You won't see sqlcmd in your SQL Server program group. Indeed, it's amazing how many people don't even know that this utility is around; that's because it's a console application rather than a Windows program.

Prior to version 7.0 and the advent of what was then called DTS (now SSIS), sqlcmd was often used in conjunction with the Bulk Copy Program (bcp) to manage the import of data from external systems. This type of use is decreasing as administrators and developers everywhere learn the power and simplicity of SSIS. Even so, there are occasionally items that you want to script into a larger command-line process. sqlcmd gives you that capability.

sqlcmd can be very handy, particularly if you use files that contain scripts. Keep in mind, however, that there are tools that can accomplish much of what sqlcmd can more effectively and with a user interface that is more consistent with the other things you're doing with your SQL Server. You can find full coverage of sqlcmd in *Professional SQL Server 2008 Programming*.

> *Once again, just for history and being able to understand if people you talk SQL Server with use a different lingo, sqlcmd is yet another new name for this tool of many names. Originally, it was referred to as ISQL. In SQL Server 2000 and 7.0, it was known as osql.*

PowerShell

PowerShell is a new feature with SQL Server 2008. PowerShell serves as an extremely robust scripting and server-navigation engine. Using PowerShell, the user can navigate all objects on the server as though they were part of a directory structure in the file system. (You even use the `dir` and `cd` style commands you use in a command window.)

PowerShell is well outside the scope of a beginning title, but it is important to realize that it's there. It is covered in more depth in *Professional SQL Server 2008 Programming*.

Summary

Most of the tools that you've been exposed to here aren't ones you'll use every day. Indeed, for the average developer, only SQL Server Management Studio will get daily use. Nevertheless it's important to have some idea of the role that each one can play. Each has something significant to offer you. We will see each of these tools again in our journey through this book.

Note that there are some other utilities available that don't have shortcuts on your Start menu (connectivity tools, server diagnostics, and maintenance utilities), which are mostly admin related.

3

The Foundation Statements of T-SQL

At last! We've finally disposed of the most boring stuff. It doesn't get any worse than basic objects and tools, does it? Unfortunately, we have to lay down a foundation before we can build a house. The nice thing is that the foundation is now down. Having used the clichéd example of building a house, I'm going to turn it all upside down by talking about the things that let you enjoy living in it before we've even talked about the plumbing. You see, when working with databases, you have to get to know how data is going to be accessed before you can learn all that much about the best ways to store it.

In this chapter, we will discuss the most fundamental *Transact-SQL (T-SQL)* statements. T-SQL is SQL Server's own dialect of Structured Query Language (SQL). The T-SQL statements that we will learn in this chapter are:

- `SELECT`
- `INSERT`
- `UPDATE`
- `DELETE`

These four statements are the bread and butter of T-SQL. We'll learn plenty of other statements as we go along, but these statements make up the basis of T-SQL's *Data Manipulation Language* (*DML)*. Because you'll generally issue far more commands meant to manipulate (that is, read and modify) data than other types of commands (such as those to grant user rights or create a table), you'll find that these will become like old friends in no time at all.

In addition, SQL provides many operators and keywords that help refine your queries. We'll learn some of the most common of these in this chapter.

While T-SQL is unique to SQL Server, the statements you use most of the time are not. T-SQL is largely ANSI/ISO compliant (The standard was originally governed by ANSI, and was later taken over by the ISO. It was ANSI long enough that people generally still refer to it as ANSI compliance.), which means that, by and large, it complies with a very wide open standard. What this means to you as a developer is that much of the SQL you're going to learn in this book is directly transferable to other SQL-based database servers such as Sybase (which long ago used to share the same code base as SQL Server), Oracle, DB2, and MySQL. Be aware, however, that every RDBMS has different extensions and performance enhancements that it uses above and beyond the ANSI/ISO standard. I will try to point out the ANSI vs. non-ANSI ways of doing things where applicable. In some cases, you'll have a choice to make — performance versus portability to other RDBMS systems. Most of the time, however, the ANSI way is as fast as any other option. In such a case, the choice should be clear: Stay ANSI compliant.

Getting Started with a Basic SELECT Statement

If you haven't used SQL before, or don't feel like you've really understood it yet, pay attention here! The `SELECT` statement and the structures used within it form the basis of the lion's share of all the commands we will perform with SQL Server. Let's look at the basic syntax rules for a `SELECT` statement:

```
SELECT [ALL|DISTINCT] [TOP (<expression>) [PERCENT] [WITH TIES]] <column list>
[FROM <source table(s)/view(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}[, XMLDATA][, ELEMENTS][, BINARY
base 64]]
[OPTION (<query hint>, [, ...n])]
```

Wow — that's a lot to decipher, so let's look at the parts.

Note that the parentheses around the `TOP` expression are, technically speaking, optional. Microsoft refers to them as "required," then points out that a lack of parentheses is actually supported, but for backward compatibility only. This means that Microsoft may pull support for that in a later release, so if you do not need to support older versions of SQL Server, I strongly recommend using parentheses to delimit a `TOP` expression in your queries.

The SELECT Statement and FROM Clause

The verb — in this case a `SELECT` — is the part of the overall statement that tells SQL Server what we are doing. A `SELECT` indicates that we are merely reading information, as opposed to modifying it. What we are selecting is identified by an expression or column list immediately following the `SELECT`. You'll see what I mean by this in a moment.

Next, we add in more specifics, such as where we are getting this data. The `FROM` statement specifies the name of the table or tables from which we are getting our data. With these, we have enough to create a

basic `SELECT` statement. Fire up the SQL Server Management Studio and let's take a look at a simple `SELECT` statement:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES;
```

Let's look at what we've asked for here. We've asked to `SELECT` information; you can also think of this as requesting to display information. The `*` may seem odd, but it actually works pretty much as `*` does everywhere: It's a wildcard. When we say `SELECT *`, we're saying we want to select every column from the table. Next, the `FROM` indicates that we've finished saying what items to output and that we're about to say what the source of the information is supposed to be — in this case, `INFORMATION_SCHEMA.TABLES`.

> `INFORMATION_SCHEMA` **is a special access path that is used for displaying metadata about your system's databases and their contents.** `INFORMATION_SCHEMA` **has several parts that can be specified after a period, such as** `INFORMATION_SCHEMA.SCHEMATA` **or** `INFORMATION_SCHEMA.VIEWS`**. These special access paths to the metadata of your system have been put there so you won't have to use system tables.**

Try It Out The SELECT Statement

Let's play around with this some more. Change the current database to be the `AdventureWorks2008` database. Recall that to do this, you need only select the `AdventureWorks2008` entry from the combo box in the toolbar at the top of the Query window in the Management Studio, as shown in Figure 3-1.

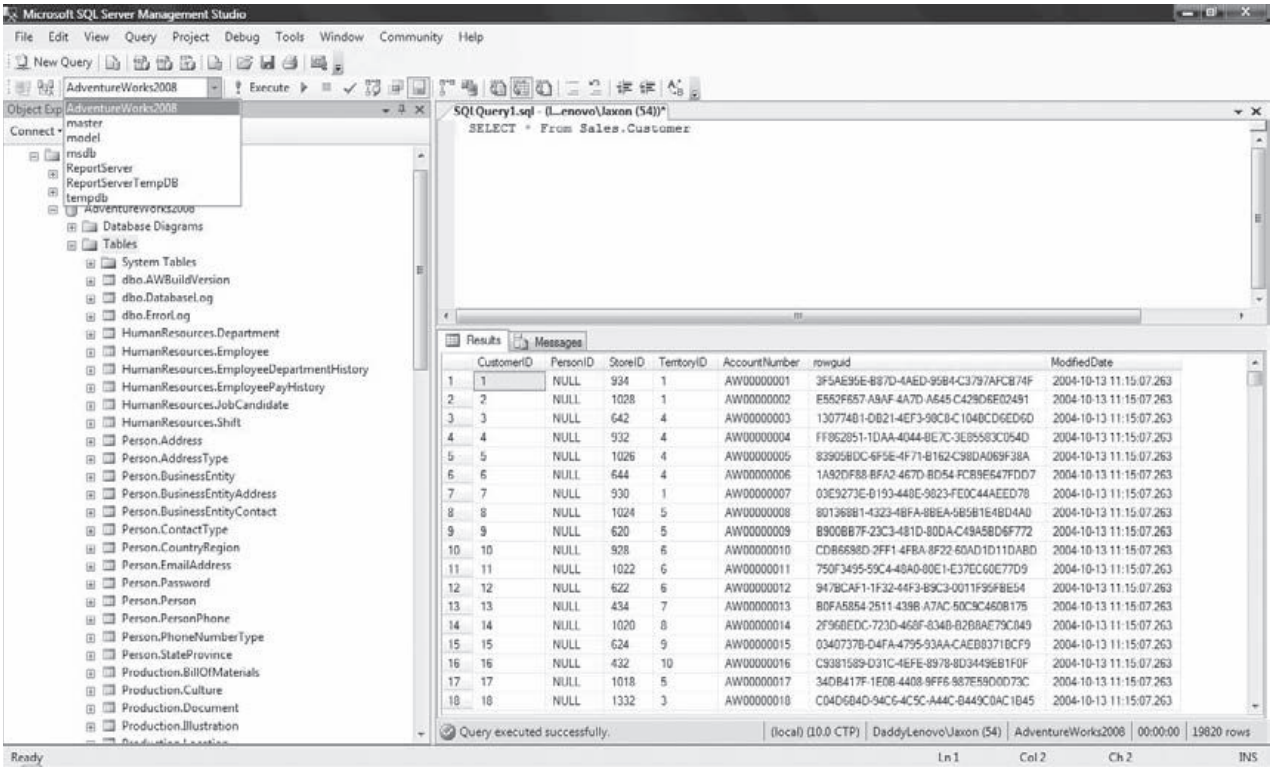

Figure 3-1

If you're having difficulty finding the combo box that lists the various databases, try clicking once in the Query window. The SQL Server Management Studio toolbars are context sensitive — that is, they change by whatever the Query window thinks is the current thing you are doing. If you don't have a Query window as the currently active window, you may have a different set of toolbars up (one that is more suitable to some other task). As soon as a Query window is active, it should switch to a set of toolbars that are suitable to query needs.

Now that we have the `AdventureWorks` database selected, let's start looking at some real data from our database. Try this query:

```
SELECT * FROM Sales.Customer;
```

After you have that in the Query window, just click Execute on the toolbar and watch SQL Server give you your results. This query will list every row of data in every column of the `Sales.Customer` table in the current database (in our case, AdventureWorks2008). If you didn't alter any of the settings on your system or the data in the AdventureWorks2008 database before you ran this query, then you should see the following information if you click on the Messages tab:

```
(19820 row(s) affected)
```

For a `SELECT` statement, the number shown here is the number of rows that your query returned. You can also find the same information on the right-hand side of the status bar (found below the results pane), with some other useful information, such as the login name of the user you're logged in as, the current database as of when the last query was run (this will persist, even if you change the database in the database drop-down box, until you run your next query in this query window), and the time it took for the query to execute.

How It Works

Let's look at a few specifics of your `SELECT` statement. Notice that I capitalized `SELECT` and `FROM`. This is not a requirement of SQL Server — we could run them as `SeLeCt` and `frOM` and they would work just fine. I capitalized them purely for purposes of convention and readability. You'll find that many SQL coders will use the convention of capitalizing all commands and keywords, while using mixed case for table, column, and non-constant variable names. The standards you choose or have forced upon you may vary, but live by at least one rule: Be consistent.

OK, time for one of my world famous soapbox diatribes. Nothing is more frustrating for a person who has to read your code or remember your table names than lack of consistency. When someone looks at your code or, more important, uses your column and table names, it shouldn't take him or her long to guess most of the way you do things just by experience with the parts that he or she has already worked with. Being consistent is one of those incredibly simple things that has been missed to at least some degree in almost every database I've ever worked with. Break the trend: Be consistent.

The `SELECT` is telling the Query window what we are doing and the `*` is saying what we want (remember that `*` = every column). Then comes the `FROM`.

A `FROM` clause does just what it says — that is, it defines the place from which our data should come. Immediately following the `FROM` will be the names of one or more tables. In our query, all of the data came from a table called `Customer`.

Now let's try taking a little bit more specific information. Let's say all we want is a list of all our customers by last name:

```
SELECT LastName FROM Person.Person;
```

Your results should look something like:

```
Achong
Abel
Abercrombie
...
He
Zheng
Hu
```

Note that I've snipped rows out of the middle for brevity. You should have 19,972 rows. Since the last name of each customer is all that we want, that's all that we've selected.

> *Many SQL writers have the habit of cutting their queries short and always selecting every column by using a * in their selection criteria. This is another one of those habits to resist. While typing in a * saves you a few moments of typing out the column names that you want, it also means that more data has to be retrieved than is really necessary. In addition, SQL Server must figure out just how many columns "*" amounts to and what specifically they are. You would be surprised at just how much this can drag down your application's performance and that of your network. In short, a good rule to live by is to select what you need — that is, exactly what you need. No more, no less.*

Let's try another simple query. How about:

```
SELECT Name FROM Production.Product;
```

Again, assuming that you haven't modified the data that came with the sample database, SQL Server should respond by returning a list of 504 different products that are available in the `AdventureWorks` database:

```
Name
--------------------------------------
Adjustable Race
Bearing Ball
BB Ball Bearing
...
...
Road-750 Black, 44
Road-750 Black, 48
Road-750 Black, 52
```

The columns that you have chosen right after your `SELECT` clause are known as the `SELECT` list. In short, the `SELECT` list is made up of the columns that you have requested be output from your query.

> **The columns that you have chosen right after your `SELECT` clause are known as the `SELECT` list.**

The WHERE Clause

Well, things are starting to get boring again, aren't they? So let's add in the `WHERE` clause. The `WHERE` clause allows you to place conditions on what is returned to you. What we have seen thus far is unrestricted information, in the sense that every row in the table specified has been included in our results. Unrestricted queries such as these are very useful for populating things like list boxes and combo boxes, and in other scenarios where you are trying to provide a *domain listing*.

> **For our purposes, don't confuse a domain with that of a Windows domain. A domain listing is an exclusive list of choices. For example, if you want someone to provide you with information about a state in the U.S., you might provide them with a list that limits the domain of choices to just the 50 states. That way, you can be sure that the option selected will be a valid one. We will see this concept of domains further when we begin talking about database design, as well as entity versus domain constraints.**

Now we want to try looking for more specific information. We don't want a listing of product names. We want information on a specific product. Try this: See if you can come up with a query that returns the name, product number, and reorder point for a product with the ProductID 356.

Let's break it down and build the query one piece at a time. First, we're asking for information to be returned, so we know that we're looking at a `SELECT` statement. Our statement of what we want indicates that we would like the product name, product number, and reorder point, so we're going to have to know what the column names are for these pieces of information. We're also going to need to know from which table or tables we can retrieve these columns.

Now we'll take a look at the tables that are available. Since we've already used the `Production.Product` table once before, we know that it's there. The `Production.Product` table has several columns. To give us a quick listing of our column options we can study the Object Explorer tree of the `Production.Product` table from Management Studio. To open this screen in the Management Studio, click Tables underneath the AdventureWorks2008 database, then expand the `Production.Product` and Columns nodes. As in Figure 3-2, you will see each of the columns along with its data type and nullability options. Again, we'll see some other methods of finding this information a little later in the chapter.

We don't have a column called `product name`, but we do have one that's probably what we're looking for: `Name`. (Original eh?) The other two columns are, save for the missing space between the two words, just as easy to identify.

Therefore, our `Products` table is going to be the place we get our information `FROM`, and the `Name`, `ProductNumber`, and `ReorderPoint` columns will be the specific columns from which we'll get our information:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
```

This query, however, still won't give us the results that we're after; it will still return too much information. Run it and you'll see that it still returns every record in the table rather than just the one we want.

If the table has only a few records and all we want to do is take a quick look at it, this might be fine. After all, we can look through a small list ourselves, right? But that's a pretty big if. In any significant system,

very few of your tables will have small record counts. You don't want to have to go scrolling through 10,000 records. What if you had 100,000 or 1,000,000? Even if you felt like scrolling through them all, the time before the results were back would be increased dramatically. Finally, what do you do when you're designing this into your application and you need a quick result that gets straight to the point?

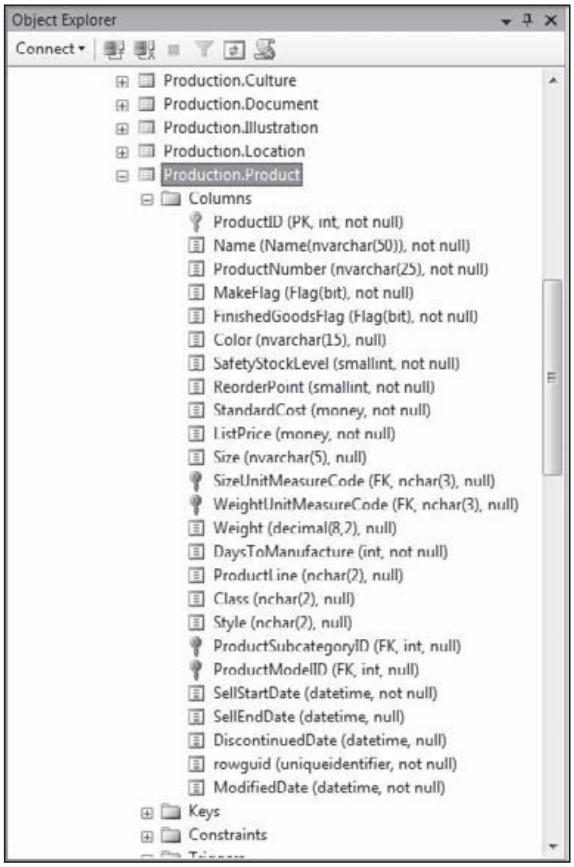

Figure 3-2

What we're after is a conditional statement that will limit the results of our query to just one product identifier — 356. That's where the `WHERE` clause comes in. The `WHERE` clause immediately follows the `FROM` clause and defines what conditions a record has to meet before it will be shown. For our query, we would want the `ProductID` to be equal to 356, so let's finish our query:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
WHERE ProductID = 356
```

Run this query against the `AdventureWorks2008` database and you should come up with:

```
Name                          ProductNumber           ReorderPoint
----------------------------  ----------------------  ------------
LL Grip Tape                  GT-0820                      600

(1 row(s) affected)
```

This time we've gotten back precisely what we wanted — nothing more, nothing less. In addition, this query runs much faster than the first query.

Let's take a look at all the operators we can use with the `WHERE` clause:

Operator	Example Usage	Effect
`=`, `>`, `<`, `>=`, `<=`, `<>`, `!=`, `!>`, `!<`	`<Column Name> = <Other Column Name>` `<Column Name> = 'Bob'`	Standard comparison operators, these work as they do in pretty much any programming language with a couple of notable points: 1. What constitutes "greater than," "less than," and "equal to" can change depending on the collation order you have selected. (For example, `"ROMEY" = "romey"` in places where case-insensitive sort order has been selected, but `"ROMEY" < > "romey"` in a case-sensitive situation.) 2. `!=` and `<>` both mean "not equal." `!<` and `!>` mean "not less than" and "not greater than," respectively.
`AND`, `OR`, `NOT`	`<Column1> = <Column2> AND <Column3> >= <Column 4>` `<Column1> != "MyLiteral" OR <Column2> = "MyOtherLiteral"`	Standard boolean logic. You can use these to combine multiple conditions into one `WHERE` clause. `NOT` is evaluated first, then `AND`, then `OR`. If you need to change the evaluation order, you can use parentheses. Note that `XOR` is not supported.
`BETWEEN`	`<Column1> BETWEEN 1 AND 5`	Comparison is `TRUE` if the first value is between the second and third values inclusive. It is the functional equivalent of `A>=B AND A<=C`. Any of the specified values can be column names, variables, or literals.
`LIKE`	`<Column1> LIKE "ROM%"`	Uses the `%` and `_` characters for wildcarding. `%` indicates a value of any length can replace the `%` character. _ indicates any one character can replace the _ character. Enclosing characters in `[ ]` symbols indicates any single character within the `[ ]` is OK. (`[a-c]` means a, b, and c are OK. `[ab]` indicates a or b are OK). ^ operates as a `NOT` operator, indicating that the next character is to be excluded.
`IN`	`<Column1> IN (List of Numbers)` `<Column1> IN ("A", "b", "345")`	Returns `TRUE` if the value to the left of the `IN` keyword matches any of the values in the list provided after the `IN` keyword. This is frequently used in subqueries, which we will look at in Chapter 16.

Operator	Example Usage	Effect
ALL, ANY, SOME	<column|expression> (comparision operator) <ANY|SOME> (subquery)	These return TRUE if any or all (depending on which you choose) values in a subquery meet the comparison operator's (e.g., <, >, =, >=) condition. ALL indicates that the value must match all the values in the set. ANY and SOME are functional equivalents and will evaluate to TRUE if the expression matches any value in the set.
EXISTS	EXISTS (subquery)	Returns TRUE if at least one row is returned by the subquery. Again, we'll look into this one further in Chapter 16.

Note that these are not the only operators in SQL Server. These are just the ones that apply to the WHERE *clause. There are a few operators that apply only as assignment operators (rather than comparison). These are inappropriate for a* WHERE *clause.*

ORDER BY

In the queries that we've run thus far, most have come out in something resembling alphabetical order. Is this by accident? It will probably come as somewhat of a surprise to you, but the answer to that is yes. If you don't say you want a specific sorting on the results of a query, then you get the data in the order that SQL Server decides to give it to you. This will always be based on what SQL Server decided was the lowest-cost way to gather the data. It will usually be based either on the physical order of a table, or on one of the indexes SQL Server used to find your data.

Microsoft's samples have a nasty habit of building themselves in a manner that happen to lend themselves to coming out in alphabetical order (long story as to why, but I wish they wouldn't do that!), but most data won't work that way. The short rendition as to why has to do with the order in which Microsoft inserts their data. Since they typically insert it in alphabetical order, the ID columns are ordered the same as alphabetical order (just by happenstance). Since most of these tables happen to be physically sorted in ID order (we'll learn more about physical sorting of the data in Chapter 9), the data wind up appearing alphabetically sorted. If Microsoft inserted the data in a more random name order — as would likely happen in a real-life scenario — then the names would tend to come out more mixed up unless you specifically asked for them to be sorted by name.

Think of an ORDER BY clause as being a sort by. It gives you the opportunity to define the order in which you want your data to come back. You can use any combination of columns in your ORDER BY clause, as long as they are columns (or derivations of columns) found in the tables within your FROM clause.

Let's look at this query:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product;
```

This will produce the following results:

```
Name                             ProductNumber             ReorderPoint
-------------------------------  ----------------          ------------
Adjustable Race                  AR-5381                        750
Bearing Ball                     BA-8327                        750
...
...
Road-750 Black, 48               BK-R19B-48                      75
Road-750 Black, 52               BK-R19B-52                      75

(504 row(s) affected)
```

As it happened, our query result set was sorted in `ProductID` order. Why? Because SQL Server decided that the best way to look at this data was by using an index that sorts the data by `ProductID`. That just happened to be what created the lowest-cost (in terms of CPU and I/O) query. Were we to run this exact query when the table has grown to a much larger size, SQL Server might have choose an entirely different execution plan, and therefore might sort the data differently. We could force this sort order by changing our query to this:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
ORDER BY Name;
```

Note that the `WHERE` clause isn't required. It can either be there or not depending on what you're trying to accomplish. Just remember that if you do have a `WHERE` clause, it goes before the `ORDER BY` clause.

Unfortunately, that previous query doesn't really give us anything different, so we don't see what's actually happening. Let's change the query to sort the data differently — by the `ProductNumber`:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
ORDER BY ProductNumber;
```

Now our results are quite different. It's the same data, but it's been substantially rearranged:

```
Name                             ProductNumber             ReorderPoint
-------------------------------  ------------------------  ------------
Adjustable Race                   AR-5381                   750
Bearing Ball                      BA-8327                   750
LL Bottom Bracket                 BB-7421                   375
ML Bottom Bracket                 BB-8107                   375
...
...
Classic Vest, L                   VE-C304-L                 3
Classic Vest, M                   VE-C304-M                 3
Classic Vest, S                   VE-C304-S                 3
Water Bottle - 30 oz.             WB-H098                   3

(504 row(s) affected)
```

SQL Server still chose the least-cost method of giving us our desired results, but the particular set of tasks it actually needed to perform changed somewhat because the nature of the query changed.

We can also do our sorting using numeric fields (note that we're querying a new table):

```
SELECT Name, SalesPersonID
FROM Sales.Store
WHERE Name BETWEEN 'g' AND 'j'
  AND SalesPersonID > 283
ORDER BY SalesPersonID, Name DESC;
```

This one results in:

```
Name                                               SalesPersonID
-------------------------------------------------- -------------
Inexpensive Parts Shop                             286
Ideal Components                                   286
Helpful Sales and Repair Service                   286
Helmets and Cycles                                 286
Global Sports Outlet                               286
Gears and Parts Company                            286
Irregulars Outlet                                  288
Hometown Riding Supplies                           288
Good Bicycle Store                                 288
Global Bike Retailers                              288
Instruments and Parts Company                      289
Instant Cycle Store                                290
Impervious Paint Company                           290
Hiatus Bike Tours                                  290
Getaway Inn                                        290

(15 row(s) affected)
```

Notice several things in this query: We've made use of many of the things that we've talked about up to this point. We've combined multiple `WHERE` clause conditions and also have an `ORDER BY` clause in place. In addition, we've added some new twists in our `ORDER BY` clause. First, we now have an `ORDER BY` clause that sorts based on more than one column. To do this, we simply comma delimited the columns we wanted to sort by. In this case, we've sorted first by `SalesPersonID` and then added a sub-sort based on `Name`. Second, the `DESC` keyword tells SQL Server that our `ORDER BY` should work in descending order for the `Name` sub-sort, rather than the default of ascending. (If you want to explicitly state that you want it to be ascending, use `ASC`.)

> **While we usually sort the results based on one of the columns that we are returning, it's worth noting that the ORDER BY clause can be based on any column in any table used in the query, regardless of whether it is included in the SELECT list.**

Aggregating Data Using the GROUP BY Clause

With ORDER BY, we have kind of taken things out of order compared with how the SELECT statement reads at the top of the chapter. Let's review the overall statement structure:

```
SELECT [TOP (<expression>) [PERCENT] [WITH TIES]] <column list>
[FROM <source table(s)/view(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}[, XMLDATA][, ELEMENTS][, BINARY
base 64]]
[OPTION (<query hint>, [, ...n])]
```

Why, if ORDER BY comes last, did we look at it before the GROUP BY? There are two reasons:

- ORDER BY is used far more often than GROUP BY, so I want you to have more practice with it.
- I want to make sure that you understand that you can mix and match all of the clauses after the FROM clause, as long as you keep them in the order that SQL Server expects them (as defined in the syntax definition).

The GROUP BY clause is used to aggregate information. Let's look at a simple query without a GROUP BY. Let's say that we want to know how many parts were ordered in a given set of orders:

```
SELECT SalesOrderID, OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672);
```

This yields a result set of:

```
SalesOrderID OrderQty
------------ --------
43660        1
43660        1
43670        1
43670        2
43670        2
43670        1
43672        6
43672        2
43672        1

(9 row(s) affected)
```

Even though we've only asked for three orders, we're seeing each individual line of detail from the orders. We can either get out our adding machine, or we can make use of the GROUP BY clause with an aggregator. In this case, we'll use SUM():

```
SELECT SalesOrderID, SUM(OrderQty)
FROM Sales.SalesOrderDetail
```

```
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

This gets us what we were looking for:

```
SalesOrderID
------------ -----------
43660        2
43670        6
43672        9

(3 row(s) affected)
```

As you would expect, the `SUM` function returns totals — but totals of what? We can easily supply an *alias* for our result. Let's modify our query slightly to provide a column name for the output:

```
SELECT SalesOrderID, SUM(OrderQty) AS TotalOrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

This gets us the same basic output, but also supplies a header to the grouped column:

```
SalesOrderID TotalOrderQty
------------ -------------
43660        2
43670        6
43672        9

(3 row(s) affected)
```

If you're just trying to get some quick results, then there really is no need to alias the grouped column as we've done here, but many of your queries are going to be written to supply information to other elements of a larger program. The code that's utilizing your queries will need some way of referencing your grouped column; aliasing your column to some useful name can be critical in that situation. We'll examine aliasing a bit more shortly.

If we didn't supply the `GROUP BY` clause, the `SUM` would have been of all the values in all of the rows for the named column. In this case, however, we did supply a `GROUP BY`, and so the total provided by the `SUM` function is the total in each group.

> **Note that when using a `GROUP BY` clause, all the columns in the `SELECT` list must either be aggregates (`SUM`, `MIN`/`MAX`, `AVG`, and so on) or columns included in the `GROUP BY` clause. Likewise, if you are using an aggregate in the `SELECT` list, your `SELECT` list must *only* contain aggregates, or there must be a `GROUP BY` clause.**

We can also group based on multiple columns. To do this we just add a comma and the next column name. Let's say, for example, that we're looking for the number of orders each salesperson has taken for

our first 10 customers. We can use both the `SalesPersonID` and `CustomerID` columns in our `GROUP BY`. (I'll explain how to use the `COUNT()` function shortly):

```
SELECT CustomerID, SalesPersonID, COUNT(*)
FROM Sales.SalesOrderHeader
WHERE CustomerID <= 11010
GROUP BY CustomerID, SalesPersonID
ORDER BY CustomerID, SalesPersonID;
```

This gets us counts, but the counts are pulled together based on how many orders a given salesperson took from a given customer:

```
CustomerID  SalesPersonID
----------- ------------- -----------
11000       NULL          3
11001       NULL          3
11002       NULL          3
11003       NULL          3
11004       NULL          3
11005       NULL          3
11006       NULL          3
11007       NULL          3
11008       NULL          3
11009       NULL          3
11010       NULL          3

 (11 row(s) affected)
```

Aggregates

When you consider that they usually get used with a `GROUP BY` clause, it's probably not surprising that aggregates are functions that work on groups of data. For example, in one of the previous queries, we got the sum of the `OrderQty` column. The sum is calculated and returned on the selected column for each group defined in the `GROUP BY` clause — in the case of our `SUM`, it was just `SalesOrderID`. A wide range of aggregates is available, but let's play with the most common.

> **While aggregates show their power when used with a `GROUP BY` clause, they are not limited to grouped queries; if you include an aggregate without a `GROUP BY`, then the aggregate will work against the entire result set (all the rows that match the `WHERE` clause). The catch here is that, when not working with a `GROUP BY`, some aggregates can only be in the `SELECT` list with other aggregates — that is, they can't be paired with a column name in the `SELECT` list unless you have a `GROUP BY`. For example, unless there is a `GROUP BY`, `AVG` can be paired with `SUM`, but not a specific column.**

AVG

This one is for computing averages. Let's try running the order quantity query we ran before, but now we'll modify it to return the average quantity per order, rather than the total for each order:

```
SELECT SalesOrderID, AVG(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

Notice that our results changed substantially:

```
SalesOrderID
------------ -------------
43660        1
43670        1
43672        3

(3 row(s) affected)
```

You can check the math — on order number 43672 there were 3 line items totaling 9 altogether (9 / 3 = 3).

MIN/MAX

Bet you can guess these two. Yes, these grab the minimum and maximum amounts for each grouping for a selected column. Again, let's use that same query modified for the `MIN` function:

```
SELECT SalesOrderID, MIN(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

Which gives the following results:

```
SalesOrderID
------------ -----------
43660        1
43670        1
43672        1

(3 row(s) affected)
```

Modify it one more time for the `MAX` function:

```
SELECT SalesOrderID, MAX(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

And you come up with this:

```
SalesOrderID
------------ -----------
43660        1
43670        2
43672        6

(3 row(s) affected)
```

What if, however, we wanted both the `MIN` and the `MAX`? Simple! Just use both in your query:

```
SELECT SalesOrderID, MIN(OrderQty), MAX(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

Now, this will yield an additional column and a bit of a problem:

```
SalesOrderID
------------ ----------- -----------
43660        1           1
43670        1           2
43672        1           6

(3 row(s) affected)
```

Can you spot the issue here? We've gotten back everything that we've asked for, but now that we have more than one aggregate column, we have a problem identifying which column is which. Sure, in this particular example we can be sure that the columns with the largest numbers are the columns generated by the `MAX` and the smallest by the `MIN`. The answer to which column is which is not always so apparent, so let's make use of an *alias*. An alias allows you to change the name of a column in the result set, and you can create it by using the `AS` keyword:

```
SELECT SalesOrderID, MIN(OrderQty) AS MinOrderQty, MAX(OrderQty) AS MaxOrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

Now our results are somewhat easier to make sense of:

```
SalesOrderID MinOrderQty MaxOrderQty
------------ ----------- -----------
43660        1           1
43670        1           2
43672        1           6

(3 row(s) affected)
```

It's worth noting that the `AS` keyword is actually optional. Indeed, there was a time (prior to version 6.5 of SQL Server) when it wasn't even a valid keyword. If you like, you can execute the same query as

before, but remove the two AS keywords from the query — you'll see that you wind up with exactly the same results. It's also worth noting that you can alias any column (and even, as we'll see in the next chapter, table names), not just aggregates.

Let's re-run this last query, but this time we'll not use the AS keyword in some places, and we'll alias every column:

```
SELECT SalesOrderID AS "Order Number", MIN(OrderQty) MinOrderQty, MAX(OrderQty)
MaxOrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID IN (43660, 43670, 43672)
GROUP BY SalesOrderID;
```

Despite the AS keyword being missing in some places, we've still changed the name output for every column:

```
Order Number MinOrderQty MaxOrderQty
------------ ----------- -----------
43660        1           1
43670        1           2
43672        1           6

(3 row(s) affected)
```

I must admit that I usually don't include the AS keyword in my aliasing, but I would also admit that it's a bad habit on my part. I've been working with SQL Server since before the AS keyword was available and have, unfortunately, become set in my ways about it (I simply forget to use it). I would, however, strongly encourage you to go ahead and make use of this extra word. Why? Well, first because it reads somewhat more clearly, and second, because it's the ANSI/ISO standard way of doing things.

So then, why did I even tell you about it? Well, I got you started doing it the right way — with the AS keyword — but I want you to be aware of alternate ways of doing things, so that you aren't confused when you see something that looks a little different.

COUNT(Expression|*)

The COUNT(*) function is about counting the rows in a query. To begin with, let's go with one of the most common varieties of queries:

```
SELECT COUNT(*)
FROM HumanResources.Employee
WHERE HumanResources.Employee.BusinessEntityID = 5;
```

The record set you get back looks a little different from what you're used to from earlier queries:

```
-----------
1

(1 row(s) affected)
```

Let's look at the differences. First, as with all columns that are returned as a result of a function call, there is no default column name. If you want there to be a column name, then you need to supply an alias. Next, you'll notice that we haven't really returned much of anything. So what does this record set represent? It is the number of rows that matched the `WHERE` condition in the query for the table(s) in the `FROM` clause.

> **Keep this query in mind. This is a basic query that you can use to verify that the exact number of rows that you expect to be in a table and match your `WHERE` condition are indeed in there.**

Just for fun, try running the query without the `WHERE` clause:

```
SELECT COUNT(*)
FROM HumanResources.Employee;
```

If you haven't done any deletions or insertions into the `Employee` table, then you should get a record set that looks something like this:

```
-----------
290

(1 row(s) affected)
```

What is that number? It's the total number of rows in the `Employee` table. This is another one to keep in mind for future use.

Now, we're just getting started! If you look back at the header for this section (the `COUNT` section), you'll see that there are two different ways of using `COUNT`. We've already discussed using `COUNT` with the * option. Now it's time to look at it with an expression — usually a column name.

First, try running the `COUNT` the old way, but against a new table:

```
SELECT COUNT(*)
FROM Person.Person;
```

This is a slightly larger table, so you get a higher `COUNT`:

```
-----------
19972

(1 row(s) affected)
```

Now alter your query to `select` the `count` for a specific column:

```
SELECT COUNT(AdditionalContactInfo)
FROM Person.Person;
```

You'll get a result that is a bit different from the one before:

```
-----
10
Warning: Null value is eliminated by an aggregate or other SET operation.

(1 row(s) affected)
```

This new result brings with it a question: why, since the `AdditionalContactInfo` column exists for every row, is there a different `COUNT` for `AdditionalContactInfo` than there is for the row count in general? The answer is fairly obvious when you stop to think about it — there isn't a value, as such, for the `AdditionalContactInfo` column in every row. In short, the `COUNT`, when used in any form other than `COUNT(*)`, ignores `NULL` values. Let's verify that `NULL` values are the cause of the discrepancy:

```
SELECT COUNT(*)
FROM Person.Person
WHERE AdditionalContactInfo IS NULL;
```

This should yield the following record set:

```
-----------
19962

(1 row(s) affected)
```

Now let's do the math:

10 + 19,962 = 19,972

That's 10 records with a defined value in the `AdditionalContactInfo` field and 19,962 rows where the value in the `AdditionalContactInfo` field is `NULL`, making a total of 19,972 rows.

> **Actually, all aggregate functions ignore `NULL`s except for `COUNT(*)`. Think about this for a minute — it can have a very significant impact on your results. Many users expect `NULL` values in numeric fields to be treated as zero when performing averages, but a `NULL` does not equal zero, and as such shouldn't be used as one. If you perform an `AVG` or other aggregate function on a column with `NULL`s, the `NULL` values will not be part of the aggregation unless you manipulate them into a non-`NULL` value inside the function (using `COALESCE()` or `ISNULL()`, for example). We'll explore this further in Chapter 7, but beware of this when coding in T-SQL and when designing your database.**
>
> **Why does it matter in your database design? Well, it can have a bearing on whether you decide to allow `NULL` values in a field or not by thinking about the way that queries are likely to be run against the database and how you want your aggregates to work.**

Before we leave the `COUNT` function, we had better see it in action with the `GROUP BY` clause.

> **For this next example, you'll need to load and execute the `BuildAndPopulateEmployee2.sql` file included with the downloadable source code (you can get that from either the `wrox.com` or `professionalsql.com` websites).**
>
> **All references to Employees in the following examples should be aimed at the new `Employees2` table rather than `Employees`.**

Let's say our boss has asked us to find out the number of employees that report to each manager. The statements that we've done thus far would either count up all the rows in the table (`COUNT(*)`) or all the rows in the table that didn't have null values (`COUNT(ColumnName)`). When we add a `GROUP BY` clause, these aggregators perform exactly as they did before, except that they return a count for each grouping rather than the full table. We can use this to get our number of reports:

```
SELECT ManagerID, COUNT(*)
FROM HumanResources.Employee2
GROUP BY ManagerID;
```

Notice that we are grouping only by the `ManagerID` — the `COUNT()` function is an aggregator and, therefore, does not have to be included in the `GROUP BY` clause.

```
ManagerID
----------- -----------
NULL        1
1           3
4           3
5           4

(4 row(s) affected)
```

Our results tell us that the manager with 1 as his/her `ManagerID` has 3 people reporting to him or her, and that 3 people report to the manager with `ManagerID` 4 as well as 4 people reporting to `ManagerID` 5. We are also able to tell that one `Employee` record had a `NULL` value in the `ManagerID` field. This employee apparently doesn't report to anyone (hmmm, president of the company I suspect?).

It's probably worth noting that we, technically speaking, could use a `GROUP BY` clause without any kind of aggregator, but this wouldn't make sense. Why not? Well, SQL Server is going to wind up doing work on all the rows in order to group them, but functionally speaking you would get the same result with a `DISTINCT` option (which we'll look at shortly), and it would operate much faster.

Now that we've seen how to operate with groups, let's move on to one of the concepts that a lot of people have problems with. Of course, after reading the next section, you'll think it's a snap.

Placing Conditions on Groups with the HAVING Clause

Up to now, all of our conditions have been against specific rows. If a given column in a row doesn't have a specific value or isn't within a range of values, then the entire row is left out. All of this happens before the groupings are really even thought about.

What if we want to place conditions on what the groups themselves look like? In other words, what if we want every row to be added to a group, but then we want to say that only after the groups are fully accumulated are we ready to apply the condition. Well, that's where the HAVING clause comes in.

The HAVING clause is used only if there is also a GROUP BY in your query. Whereas the WHERE clause is applied to each row before it even has a chance to become part of a group, the HAVING clause is applied to the aggregated value for that group.

Let's start off with a slight modification to the GROUP BY query we used at the end of the previous section — the one that tells us the number of employees assigned to each manager's EmployeeID:

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
GROUP BY ManagerID;
```

In the next chapter, we'll learn how to put names on the EmployeeIDs that are in the Manager column. For now though, we'll just note that there appear to be three different managers in the company. Apparently, everyone reports to these three people, except for one person who doesn't have a manager assigned — that is probably our company president (we could write a query to verify that, but we'll just trust in our assumptions for now).

We didn't put a WHERE clause in this query, so the GROUP BY was operating on every row in the table and every row is included in a grouping. To test what would happen to our COUNTs, let's add a WHERE clause:

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
WHERE EmployeeID != 5
GROUP BY ManagerID;
```

This yields one slight change that may be somewhat different than expected:

```
Manager     Reports
----------- -----------
NULL        1
1           3
4           2
5           4

(4 row(s) affected)
```

No rows were eliminated from the result set, but the result for ManagerID 4 was decreased by one (what the heck does this have to do with ManagerID 5?). You see, the WHERE clause eliminated the one row where the EmployeeID was 5. As it happens, EmployeeID 5 reports to ManagerID 4, so the total for ManagerID 4 was one less (EmployeeID 5 is no longer counted for this query). ManagerID 5 was not affected, as we eliminated him or her as a report (as an EmployeeID) rather than as a manager. The key thing here is to realize that EmployeeID 5 was eliminated *before* the GROUP BY was applied.

I want to look at things a bit differently though. See if you can work out how to answer the following question. Which managers have more than three people reporting to them? You can look at the query without the WHERE clause and tell by the COUNT, but how do you tell programmatically? That is, what if we need this query to return only the managers with more than three people reporting to them? If you try to work this out with a WHERE clause, you'll find that there isn't a way to return rows based on the

aggregation. The WHERE clause is already completed by the system before the aggregation is executed. That's where our HAVING clause comes in:

```
SELECT ManagerID AS Manager, COUNT(*) AS Reports
FROM HumanResources.Employee2
WHERE EmployeeID != 5
GROUP BY ManagerID
HAVING COUNT(*) > 3;
```

Try it out and you'll come up with something a little bit more like what we were after:

```
Manager     Reports
----------- -----------
5           4

(1 row(s) affected)
```

There is only one manager that has more than three employees reporting to him or her.

Outputting XML Using the FOR XML Clause

SQL Server has a number of features to natively support XML. From being able to index XML data effectively to validating XML against a schema document, SQL Server is very robust in meeting XML data storage and manipulation needs.

One of the oldest features in SQL Server's XML support arsenal is the FOR XML clause you can use with the SELECT statement. Use of this clause causes your query output to be supplied in an XML format, and a number of options are available to allow fairly specific control of exactly how that XML output is styled. I'm going to shy away from the details of this clause for now, since XML is a discussion unto itself, but we'll spend extra time with XML in Chapter 16. So for now, just trust me that it's better to learn the basics first.

Making Use of Hints Using the OPTION Clause

The OPTION clause is a way of overriding some of SQL Server's ideas of how best to run your query. Since SQL Server really does usually know what's best for your query, using the OPTION clause will more often hurt you than help you. Still, it's nice to know that it's there, just in case.

This is another one of those "I'll get there later" subjects. We talk about query hints extensively when we talk about locking later in the book, but until you understand what you're affecting with your hints, there is little basis for understanding the OPTION clause. As such, we'll defer discussion of it for now.

The DISTINCT and ALL Predicates

There's just one more major concept to get through and we'll be ready to move from the SELECT statement on to action statements. It has to do with repeated data.

Let's say, for example, that we wanted a list of the IDs for all of the products that we have sold at least 30 of in an individual sale (more than 30 at one time). We can easily get that information from the SalesOrderDetail table with the following query:

```
SELECT ProductID
FROM Sales.SalesOrderDetail
WHERE OrderQty > 30;
```

What we get back is one row matching the ProductID for every row in the SalesOrderDetail table that has an order quantity that is more than 30:

```
ProductID
-----------
709
863
863
863
863
863
863
863
715
863
...
...
869
869
867

(31 row(s) affected)
```

While this meets your needs from a technical standpoint, it doesn't really meet your needs from a reality standpoint. Look at all those duplicate rows! While we could look through and see which products sold more than 30 at a time, the number of rows returned and the number of duplicates can quickly become overwhelming. Like the problems we've discussed before, we have an answer. It comes in the form of the DISTINCT predicate on your SELECT statement.

Try re-running the query with a slight change:

```
SELECT DISTINCT ProductID
FROM Sales.SalesOrderDetail
WHERE OrderQty > 30;
```

Now you come up with a true list of the ProductIDs that sold more than 30 at one time:

```
ProductID
-----------
863
869
709
864
867
715

(6 row(s) affected)
```

As you can see, this cut down the size of your list substantially and made the contents of the list more relevant. Another side benefit of this query is that it will actually perform better than the first one. Why? Well, I go into that later in the book when I discuss performance issues further, but for now, suffice it to say that not having to return every single row means that SQL Server doesn't have to do quite as much work in order to meet the needs of this query.

As the old commercials on television go, "But wait! There's more!" We're not done with `DISTINCT` yet. Indeed, the next example is one that you might be able to use as a party trick to impress your programmer friends. You see, this is one that an amazing number of SQL programmers don't even realize you can do. `DISTINCT` can be used as more than just a predicate for a `SELECT` statement. It can also be used in the expression for an aggregate. What do I mean? Let's compare three queries.

First, grab a row count for the `SalesOrderDetail` table in AdventureWorks:

```
SELECT COUNT(*)
FROM Sales.SalesOrderDetail;
```

If you haven't modified the `SalesOrderDetail` table, this should yield you around 121,317 rows.

Now run the same query using a specific column to `COUNT`:

```
SELECT COUNT(SalesOrderID)
FROM Sales.SalesOrderDetail;
```

Since the `SalesOrderID` column is part of the key for this table, it can't contain any `NULL`s (more on this in the chapter in indexing). Therefore, the net count for this query is always going to be the same as the `COUNT(*)` — in this case, it's 121,317.

> ***Key* is a term used to describe a column or combination of columns that can be used to identify a row within a table. There are actually several different kinds of keys (we'll see much more on these in Chapters 6, 8, and 9), but when the word "key" is used by itself, it is usually referring to a table's primary key. A primary key is a column (or group of columns) that is (are) effectively the unique name for that row. When you refer to a row using its primary key, you can be certain that you will get back only one row, because no two rows are allowed to have the same primary key within the same table.**

Now for the fun part. Modify the query again:

```
SELECT COUNT(DISTINCT SalesOrderID)
FROM Sales.SalesOrderDetail;
```

Now we get a substantially different result:

```
-----------
31465

(1 row(s) affected)
```

All duplicate rows were eliminated before the aggregation occurred, so you have substantially fewer rows.

> *Note that you can use* DISTINCT *with any aggregate function, although I question whether many of the functions have any practical use for it. For example, I can't imagine why you would want an average of just the* DISTINCT *rows.*

That takes us to the ALL predicate. With one exception, it is a very rare thing indeed to see someone actually including an ALL in a statement. ALL is perhaps best understood as being the opposite of DISTINCT. Where DISTINCT is used to filter out duplicate rows, ALL says to include every row. ALL is the default for any SELECT statement, except for situations where there is a UNION. We will discuss the impact of ALL in a UNION situation in the next chapter, but for now, realize that ALL is happening any time you don't ask for a DISTINCT.

Adding Data with the INSERT Statement

By now you should pretty much have the hang of basic SELECT statements. We would be doing well to stop here, save for a pretty major problem. We wouldn't have very much data to look at if we didn't have some way of getting it into the database in the first place. That's where the INSERT statement comes in.

The full syntax for INSERT has several parts:

```
INSERT [TOP ( <expression> ) [PERCENT] ] [INTO] <tabular object>
  [(<column list>)]
  [ OUTPUT <output clause> ]
{ VALUES (<data values>) [,(<data values>)] [, …n]
   | <table source>
   | EXEC <prodecure>
   | DEFAULT VALUES
```

This is a bit wordy to worry about now, so let's simplify it a bit. The more basic syntax for an INSERT statement looks like this:

```
INSERT [INTO] <table>
  [(<column list>)]
VALUES (<data values>) [,(<data values>)] [, …n]
```

Let's look at the parts.

INSERT is the action statement. It tells SQL Server what it is that we're going to be doing with this statement; everything that comes after this keyword is merely spelling out the details of that action.

The INTO keyword is pretty much just fluff. Its sole purpose in life is to make the overall statement more readable. It is completely optional, but I highly recommend its use for the very reason that they added it to the statement: It makes things much easier to read. As we go through this section, try a few of the statements both with and without the INTO keyword. It's a little less typing if you leave it out, but it's also quite a bit stranger to read — it's up to you.

Next comes the table (technically table, view, or a common table expression, but we're just going to worry about tables for now) into which you are inserting.

Until this point, things have been pretty straightforward. Now comes the part that's a little more difficult: the column list. An explicit column list (where you specifically state the columns to receive values) is optional, but not supplying one means that you have to be extremely careful. If you don't provide an explicit column list, then each value in your `INSERT` statement will be assumed to match up with a column in the same ordinal position of the table in order (first value to first column, second value to second column, and so on). Additionally, a value must be supplied for every column, in order, until you reach the last column that both does not accept `NULL`s and has no default. (You'll see more about what I mean shortly.) The exception is an `IDENTITY` column, which should be skipped when supplying values (SQL Server will fill that in for you). In summary, this will be a list of one or more columns that you are going to be providing data for in the next part of the statement.

Finally, you'll supply the values to be inserted. There are two ways of doing this, but for now, we'll focus on single line inserts that use data that you explicitly provide. To supply the values, we'll start with the `VALUES` keyword, then follow that with a list of values separated by commas and enclosed in parentheses. The number of items in the value list must exactly match the number of columns in the column list. The data type of each value must match or be implicitly convertible to the type of the column with which it corresponds (they are taken in order).

> *On the issue of whether to specifically state a value for all columns or not, I really recommend naming every column every time, even if you just use the* `DEFAULT` *keyword or explicitly state* `NULL`*.* `DEFAULT` *will tell SQL Server to use whatever the default value is for that column (if there isn't one, you'll get an error).*

What's nice about this is the readability of code; this way it's really clear what you are doing. In addition, I find that explicitly addressing every column leads to fewer bugs.

Whew! That's confusing, so let's practice with this some.

To get started with `INSERT`s, `UPDATE`s, and `DELETE`s, we're going to create a couple of tables to work with. (`AdventureWorks` is a bit too bulky for just starting out.) To be ready to try these next few examples, you'll need to run a few statements that we haven't really discussed as yet. Try not to worry about the contents of this yet; we'll get to discussing them fully in Chapter 5.

You can either type these and execute them, or use the file called `Chap3CreateExampleTables.sql` included with the downloadable files for this book.

> **This next block of code is what is called a script. This particular script is made up of one batch. We will be examining batches at length in Chapter 11.**

```
/* This script creates a couple of tables for use with
** several examples in Chapter 3 of Beginning SQL Server
** 2008 Programming
*/

CREATE TABLE Stores
(
  StoreCode    char(4)      NOT NULL PRIMARY KEY,
  Name         varchar(40)  NOT NULL,
  Address      varchar(40)  NULL,
```

```
    City          varchar(20)  NOT NULL,
    State         char(2)      NOT NULL,
    Zip           char(5)      NOT NULL
  );

  CREATE TABLE Sales
  (
    OrderNumber   varchar(20)  NOT NULL PRIMARY KEY,
    StoreCode     char(4)      NOT NULL
      FOREIGN KEY REFERENCES Stores(StoreCode),
    OrderDate     date         NOT NULL,
    Quantity      int          NOT NULL,
    Terms         varchar(12)  NOT NULL,
    TitleID       int          NOT NULL
  );
```

Most of the inserts we're going to do in this chapter will be to the `Stores` table we just created, so let's review the properties for that table. To do this, expand the Tables node of whichever database was current when you ran the preceding script (probably `AdventureWorks` given the other examples we've been running) in the Object Explorer within the Management Studio. Then expand the Columns node as shown in Figure 3-3.

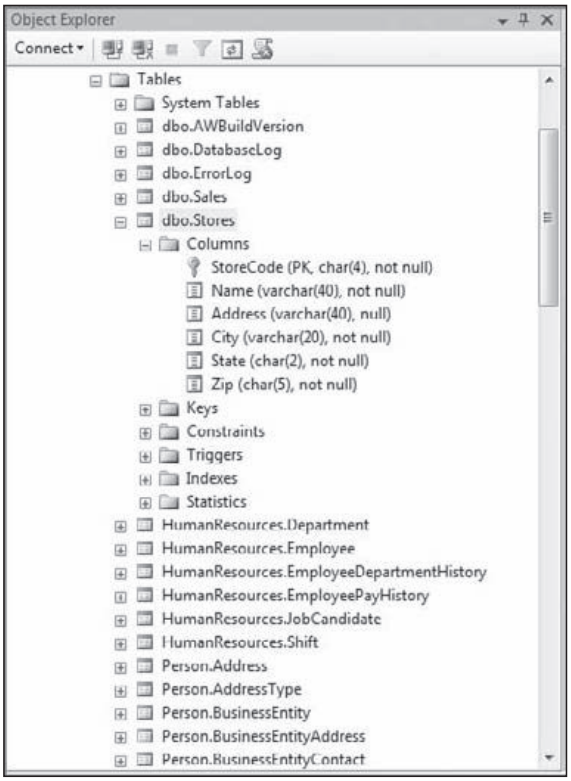

Figure 3-3

In this table, every column happens to be a `char` or `varchar`.

For your first insert, we'll eliminate the optional column list and allow SQL Server to assume we're providing something for every column:

```
INSERT INTO Stores
VALUES
   ('TEST', 'Test Store', '1234 Anywhere Street', 'Here', 'NY', '00319');
```

As stated earlier, unless we provide a different column list (we'll cover how to provide a column list shortly), all the values have to be supplied in the same order as the columns are defined in the table. After executing this query, you should see a statement that tells you that one row was affected by your query. Now, just for fun, try running the exact same query a second time. You'll get the following error:

```
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__Stores__1387E197'. Cannot insert duplicate
key in object 'dbo.Stores'.
The statement has been terminated.
```

Why did it work the first time and not the second? Because this table has a primary key that does not allow duplicate values for the `StoreCode` field. As long as we changed that one field, we could have left the rest of the columns alone and it would have taken the new row. We'll see more of primary keys in the chapters on design and constraints.

So let's see what we inserted:

```
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

This query yields us exactly what we inserted:

```
StoreCode Name         Address                 City                State Zip
--------- ------------ ----------------------- ------------------- ----- -----
TEST      Test Store   1234 Anywhere Street    Here                NY    00319

(1 row(s) affected)
```

Note that I've trimmed a few spaces off the end of each column to help it fit on a page neatly, but the true data is just as we expected it to be.

Now let's try it again with modifications for inserting into specific columns:

```
INSERT INTO Stores
   (StoreCode, Name, City, State, Zip)
VALUES
   ('TST2', 'Test Store', 'Here', 'NY', '00319');
```

Note that on the line with the data values we've changed just two things. First, we've changed the value we are inserting into the primary key column so it won't generate an error. Second, we've eliminated the value that was associated with the `Address` column, since we have omitted that column in our column list. There are a few different instances where we can skip a column in a column list and not provide any data

for it in the INSERT statement. For now, we're just taking advantage of the fact that the Address column is not a required column — that is, it accepts NULLs. Since we're not providing a value for this column and since it has no default (we'll see more on defaults later on), this column will be set to NULL when we perform our INSERT. Let's verify that by rerunning our test SELECT statement with one slight modification:

```
SELECT *
FROM Stores
WHERE StoreCode = 'TST2';
```

Now we see something a little different:

```
StoreCode    Name            Address           City      State      Zip
---------    -----------     ---------------   -------   --------   ------
TST2         Test Store      NULL              Here      NY         00319

(1 row(s) affected)
```

Notice that a NULL was inserted for the column that we skipped.

Note that the columns have to be *nullable* in order to do this. What does that mean? Pretty much what it sounds like: It means that you are allowed to have NULL values for that column. Believe me, we will be discussing the nullability of columns at great length in this book, but for now, just realize that some columns allow NULLs and some don't. We can always skip providing information for columns that allow NULLs.

If, however, the column is not nullable, then one of three conditions must exist, or we will receive an error and the INSERT will be rejected:

- The column has been defined with a *default value*. A default is a constant value that is inserted if no other value is provided. We will learn how to define defaults in Chapter 7.
- The column is defined to receive some form of system-generated value. The most common of these is an IDENTITY value (covered more in the design chapter), where the system typically starts counting first at one row, increments to two for the second, and so on. These aren't really row numbers, as rows may be deleted later on and numbers can, under some conditions, get skipped, but they serve to make sure each row has its own identifier.
- We supply a value for the column.

Just for completeness, let's perform one more INSERT statement. This time, we'll insert a new sale into the Sales table. To view the properties of the Sales table, we can either open its Properties dialog as we did with the Stores table, or we can run a system-stored procedure called sp_help. sp_help will report information about any database object, user-defined data type, or SQL Server data type. The syntax for using sp_help is as follows:

```
EXEC sp_help <name>
```

To view the properties of the Sales table, we just have to type the following into the Query window:

```
EXEC sp_help Sales;
```

Which returns (among other things):

```
Column_name             Type        Computed    Length      Prec  Scale Nullable
----------------------- ----------- ----------- ----------- ----- ----- --------
OrderNumber             varchar     no          20                      no
StoreCode               char        no          4                       no
OrderDate               date        no          3           10    0     no
Quantity                int         no          4           10    0     no
Terms                   varchar     no          12                      no
TitleID                 int         no          4           10    0     no
```

The `Sales` table has six columns in it, but pay particular attention to the `Quantity`, `OrderDate`, and `TitleID` columns; they are of types that we haven't done `INSERT`s with up to this point.

What you need to pay attention to in this query is how to format the types as you're inserting them. We do *not* use quotes for numeric values as we have with our character data. However, the `date` data type does require quotes. (Essentially, it goes in as a string and it then gets converted to a date.)

```
INSERT INTO Sales
   (StoreCode, OrderNumber, OrderDate, Quantity, Terms, TitleID)
VALUES
   ('TEST', 'TESTORDER', '01/01/1999', 10, 'NET 30', 1234567);
```

This gets back the now familiar `(1 row(s) affected)` message. Notice, however, that I moved around the column order in my insert. The data in the `VALUES` portion of the insert needs to match the column list, but the column list can be in any order you choose to make it; it does not have to be in the order the columns are listed in the physical table.

Note that while I've used the MM/DD/YYYY format that is popular in the U.S., you can use a wide variety of other formats (such as the internationally more popular YYYY-MM-DD) with equal success. The default for your server will vary depending on whether you purchase a localized copy of SQL Server or if the setting has been changed on the server.

Multirow Inserts

New with SQL Server 2008 is the ability to `INSERT` multiple rows at one time. To do this, just keep tacking on additional comma-delimited insertion values, for example:

```
INSERT INTO Sales
   (StoreCode, OrderNumber, OrderDate, Quantity, Terms, TitleID)
VALUES
   ('TST2', 'TESTORDER2', '01/01/1999', 10, 'NET 30', 1234567),
   ('TST2', 'TESTORDER3', '02/01/1999', 10, 'NET 30', 1234567);
```

This inserts both sets of values using a single statement. To check what we got, let's go ahead and look in the `Sales` table:

```
SELECT *
FROM Sales;
```

And sure enough, we get back the one row we inserted earlier along with the two rows we just inserted:

```
OrderNumber          StoreCode OrderDate  Quantity    Terms        TitleID
-------------------- --------- ---------- ----------- ------------ -----------
TESTORDER            TEST      1999-01-01 10          NET 30       1234567
TESTORDER2           TST2      1999-01-01 10          NET 30       1234567
TESTORDER3           TST2      1999-02-01 10          NET 30       1234567

(3 row(s) affected)
```

This new feature has the potential to really boost performance in situations where you are performing multiple `INSERT`*s. Previously, your client application would have to issue a completely separate* `INSERT` *statement for each row of data you wanted to* `INSERT` *(there were some ways around this, but they required extra thought and effort that it seems few developers were willing to put in). Using this method can eliminate many round trips to your server; just keep in mind that it also means that your application will not be backward compatible to prior versions of SQL Server.*

The INSERT INTO . . . SELECT Statement

What if we have a block of data that we want `INSERT`ed? As we have just seen, we can perform multi-row `INSERT`s explicitly, but what if we want to `INSERT` a block of data that can be selected from another source, such as:

- Another table in your database
- A totally different database on the same server
- A heterogeneous query from another SQL Server or other data
- The same table (usually you're doing some sort of math or other adjustment in your `SELECT` statement, in this case)

The `INSERT INTO . . . SELECT` statement can `INSERT` data from any of these. The syntax for this statement comes from a combination of the two statements we've seen thus far — the `INSERT` statement and the `SELECT` statement. It looks something like this:

```
INSERT INTO <table name>
[<column list>]
<SELECT statement>
```

The result set created from the `SELECT` statement becomes the data that is added in your `INSERT` statement.

Let's check this out by doing something that, if you get into advanced coding, you'll find yourself doing all too often — `SELECT`ing some data into some form of temporary table. In this case, we're going to declare a variable of type `table` and fill it with rows of data from our `Orders` table:

> **Like our early example database creation, this next block of code is what is called a script. Again, we will be examining batches at length in Chapter 11.**

```
/* This next statement is going to use code to change the "current" database
** to AdventureWorks2008. This makes certain, right in the code that we are going
** to the correct database.
*/

USE AdventureWorks2008;

/* This next statement declares our working table.
** This particular table is actually a variable we are declaring on the fly.
*/

DECLARE @MyTable Table
(
   SalesOrderID      int,
   CustomerID        char(5)
);

/* Now that we have our table variable, we're ready to populate it with data
** from our SELECT statement. Note that we could just as easily insert the
** data into a permanent table (instead of a table variable).
*/
INSERT INTO @MyTable
   SELECT SalesOrderID, CustomerID
   FROM AdventureWorks2008.Sales.SalesOrderHeader
   WHERE SalesOrderID BETWEEN 44000 AND 44010;

-- Finally, let's make sure that the data was inserted like we think
SELECT *
FROM @MyTable;
```

This should yield you results that look like this:

```
(11 row(s) affected)
SalesOrderID CustomerID
------------ ----------
44000        27918
44001        28044
44002        14572
44003        19325
44004        28061
44005        26629
44006        16744
44007        25555
44008        27678
44009        27759
44010        13680

(11 row(s) affected)
```

The first instance of `(11 row(s) affected)` we see is the effect of the `INSERT...SELECT` statement at work. Our `SELECT` statement returned three rows, so that's what got `INSERT`ed into our table. We then used a straight `SELECT` statement to verify the `INSERT`.

> **Note that if you try running a `SELECT` against `@MyTable` by itself (that is, outside this script), you're going to get an error. `@MyTable` is a declared variable and it exists only as long as our batch is running. After that, it is automatically destroyed.**
>
> **It's also worth noting that we could have used what's called a *temporary table*. This is similar in nature, but doesn't work in quite the same way. We will revisit temp tables and table variables in Chapters 11 through 13.**

Changing What You've Got with the UPDATE Statement

The `UPDATE` statement, like most SQL statements, does pretty much what it sounds like it does — it updates existing data. The structure is a little bit different from a `SELECT`, although you'll notice definite similarities. Like the `INSERT` statement, it has a fairly complex set of options, but a more basic version that will meet the vast majority of your needs.

The full syntax supports the same `TOP` and similar predicates that were supported under `SELECT` and `INSERT`:

```
UPDATE [TOP ( <expression> ) [PERCENT] ] <tabular object>
  SET <column> = <value>[.WRITE(<expression>, <offset>, <length>)]
    [,<column> = <value>[.WRITE(<expression>, <offset>, <length>)]]
  [ OUTPUT <output clause> ]
[FROM <source table(s)>]
[WHERE <restrictive condition>]
```

Let's look at the more basic syntax:

```
UPDATE <table name>
SET <column> = <value> [,<column> = <value>]
[FROM <source table(s)>]
[WHERE <restrictive condition>]
```

An `UPDATE` can be created from multiple tables, but can affect only one table. What do I mean by that? Well, we can build a condition, or retrieve values from any number of different tables, but only one table at a time can be the subject of the `UPDATE` action. Don't sweat this one too much. We haven't looked at joining multiple tables yet (next chapter folks!), so we won't get into complex `UPDATE` statements here. For now, we'll look at simple updates.

Let's start by doing some updates to the data that we inserted in the `INSERT` statement section. Let's rerun that query to look at one row of inserted data. (Don't forget to switch back to the pubs database.):

```
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

Which returns the following:

```
StoreCode      Name           Address                City       State       Zip
----------     ------------   --------------------   -------    ---------   ----
TEST           Test Store     1234 Anywhere Street   Here       NY          00319
```

Let's update the value in the `City` column:

```
UPDATE Stores
SET City = 'There'
WHERE StoreCode = 'TEST';
```

Much like when we ran the `INSERT` statement, we don't get much back from SQL Server:

```
(1 row(s) affected)
```

Yet when we run our `SELECT` statement again, we see that the value has indeed changed:

```
StoreCode      Name           Address                City       State       Zip
----------     ------------   --------------------   -------    ---------   ----
TEST           Test Store     1234 Anywhere Street   There       NY          00319
```

Note that we could have changed more than one column just by adding a comma and the additional column expression. For example, the following statement would have updated both columns:

```
UPDATE Stores
SET city = 'There', state = 'NY'
WHERE StoreCode = 'TEST';
```

If we choose, we can use an expression for the `SET` clause instead of the explicit values we've used thus far. For example, we could add a suffix on all of the store names by concatenating the existing store name with a suffix:

```
UPDATE Stores
SET Name = Name + ' - ' + StoreCode;
```

After executing that `UPDATE`, run a `SELECT` statement on Stores:

```
SELECT *
FROM Stores
```

You should see the `Name` suffixed by the `StoreCode`:

```
StoreCode   Name                Address                City       State    Zip
----------  --------------      --------------------   -------    ------   ----
TEST        Test Store - TEST   1234 Anywhere Street   There      NY       00319
TST2        Test Store - TST2   NULL                   Here       NY       00319
```

As you can see, a single `UPDATE` statement can be fairly powerful. Even so, this is really just the beginning. We'll see even more advanced updates in later chapters.

While SQL Server is nice enough to let us `UPDATE` pretty much any column (there are a few that we can't, such as `timestamp`s), be very careful about updating primary keys. Doing so puts you at very high risk of *orphaning* other data (data that has a reference to the data you're changing).

For example, the `StoreCode` field in the `Stores` table is a primary key. If we decide to change `StoreCode` 10 to 35 in `Stores`, then any data in the `Sales` table that relates to that store may be orphaned and lost if the `StoreCode` value in all of the records relating to `StoreCode` 10 is not also updated to 35. As it happens, there is a constraint that references the `Stores` table, so SQL Server would prevent such an orphaning situation in this case (we'll investigate constraints in Chapter 7), but updating primary keys is risky at best.

The DELETE Statement

The version of the `DELETE` statement that we'll cover in this chapter may be one of the easiest statements of them all. Even in the more complex syntax, there's no column list, just a table name and (usually) a `WHERE` clause. The full version looks like this:

```
DELETE [TOP ( <expression> ) [PERCENT] ] [FROM] <tabular object>
  [ OUTPUT <output clause> ]
[FROM <table or join condition>]
[WHERE <search condition> | CURRENT OF [GLOBAL] <cursor name>]
```

The basic syntax couldn't be much easier:

```
DELETE <table name>
[WHERE <condition>]
```

The `WHERE` clause works just like all the `WHERE` clauses we've seen thus far. We don't need to provide a column list because we are deleting the entire row. (You can't delete half a row, for example.)

Because this is so easy, we'll perform only a couple of quick `DELETE`s that are focused on cleaning up the `INSERT`s that we performed earlier in the chapter. First, let's run a `SELECT` to make sure the first of those rows is still there:

```
SELECT *
FROM Stores
WHERE StoreCode = 'TEST';
```

If you haven't already deleted it, you should come up with a single row that matches what we added with our original `INSERT` statement. Now let's get rid of it:

```
DELETE Stores
WHERE StoreCode = 'TEST';
```

Note that we've run into a situation where SQL Server is refusing to DELETE this row because of referential integrity violations:

```
Msg 547, Level 16, State 0, Line 1
The DELETE statement conflicted with the REFERENCE constraint
"FK__Sales__StoreCode__1B29035F". The conflict occurred in database
"AdventureWorks", table "dbo.Sales", column 'StoreCode'.
The statement has been terminated.
```

SQL Server won't let us DELETE a row if it is referenced as part of a foreign key constraint. We'll see much more on foreign keys in Chapter 7, but for now, just keep in mind that if one row references another row (either in the same or a different table — it doesn't matter) using a foreign key, then the referencing row must be deleted before the referenced row can be deleted. One of our INSERT statements inserted a record into the Sales table that had a StoreCode of TEST — this record is referencing the record we have just attempted to DELETE.

Before we can delete the record from our Stores table, we must delete the record it is referencing in the Sales table:

```
DELETE Sales
WHERE StoreCode = 'TEST';
```

Now we can successfully rerun the first DELETE statement:

```
DELETE Stores
WHERE StoreCode_id = 'TEST';
```

You can do two quick checks to verify that the data was indeed deleted. The first happens automatically when the DELETE statement is executed; you should get a message telling you that one row was affected. The other quick check is to rerun the SELECT statement; you should get zero rows back.

For one more easy practice DELETE, we'll also kill that second row by making just a slight change:

```
DELETE Sales
WHERE StoreCode = 'TST2';
```

That's it for simple DELETEs! Like the other statements in this chapter, we'll come back to the DELETE statement when we're ready for more complex search conditions.

Summary

T-SQL is SQL Server's own brand of ANSI/ISO SQL or Structured Query Language. T-SQL is largely ANSI/ISO compliant, but it also has a number of its own extensions to the language. We'll see more of those in later chapters.

Even though, for backward compatibility, SQL Server has a number of different syntax choices that are effectively the same, wherever possible you ought to use the ANSI form. Where there are different choices available, I will usually show you all of the choices, but again, stick with the ANSI/ISO version wherever possible. This is particularly important for situations where you think your backend — or database

server — might change at some point. Your ANSI code will more than likely run on the new database server; however, code that is only T-SQL definitely will not.

In this chapter, you have gained a solid taste of making use of single table statements in T-SQL, but the reality is that you often need information from more than one table. In the next chapter, we will learn how to make use of `JOIN`s to allow us to use multiple tables.

Exercises

1. Write a query that outputs all of the columns and all of the rows from the `Product` table (in the `Production` schema) of the pubs database.
2. Modify the query in Exercise 1 so it filters down the result to just the products that have no `ProductSubcategoryID`. (HINT: There are 209, and you will need to be looking for `NULL` values.)
3. Add a new row into the `ProductLocation` (in the `Production` schema) table in the AdventureWorks database.
4. Remove the row you just added.

4

JOINs

Feel like a seasoned professional yet? Let me dash that feeling right away (just kidding)! While we now have the basic statements under our belt, they are only a small part of the bigger picture of the statements we will run. To put it simply, there is often not that much you can do with just one table — especially in a highly normalized database.

A *normalized* database is one where the data has been broken out from larger tables into many smaller tables for the purpose of eliminating repeating data, saving space, improving performance, and increasing data integrity. It's great stuff and vital to relational databases; however, it also means that you wind up getting your data from here, there, and everywhere.

> *We will be looking into the concepts of normalization extensively in Chapter 8. For now, though, just keep in mind that the more normalized your database is, the more likely that you're going to have to join multiple tables together in order to get all the data you want.*

In this chapter, I'm going to introduce you to the process of combining tables into one result set by using the various forms of the `JOIN` clause. These will include:

- `INNER JOIN`
- `OUTER JOIN` (both `LEFT` and `RIGHT`)
- `FULL JOIN`
- `CROSS JOIN`

We'll also learn that there is more than one syntax available to use for joins, and that one particular syntax is the right choice. In addition, we'll take a look at the `UNION` operator, which allows us to combine the results of two queries into one.

JOINs

When we are operating in a normalized environment, we frequently run into situations in which not all of the information that we want is in one table. In other cases, all the information we want

returned is in one table, but the information we want to place conditions on is in another table. This is where the JOIN clause comes in.

A JOIN does just what it sounds like — it puts the information from two tables together into one result set. We can think of a result set as being a "virtual" table. It has both columns and rows, and the columns have data types. Indeed, in Chapter 7, we'll see how to treat a result set as if it were a table and use it for other queries.

How exactly does a JOIN put the information from two tables into a single result set? Well, that depends on how you tell it to put the data together — that's why there are four different kinds of JOINs. The thing that all JOINs have in common is that they match one record up with one or more other records to make a record that is a superset created by the combined columns of both records.

For example, let's take a record from a table we'll call Films:

FilmID	FilmName	YearMade
1	My Fair Lady	1964

Now let's follow that up with a record from a table called Actors:

FilmID	FirstName	LastName
1	Rex	Harrison

With a JOIN, we could create one record from two records found in totally separate tables:

ilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison

This JOIN (at least apparently) joins records in a one-to-one relationship. We have one Films record joining to one Actors record.

Let's expand things just a bit and see if you can see what's happening. I've added another record to the Actors table:

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn

Now let's see what happens when we join that to the very same (only one record) `Films` table:

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn

As you can see, the result has changed a bit — we are no longer seeing things as being one-to-one, but rather one-to-two, or more appropriately, what we would call one-to-many. We can use that single record in the `Films` table as many times as necessary to have complete (joined) information about the matching records in the `Actors` table.

Have you noticed how they are matching up? It is, of course, by matching up the `FilmID` field from the two tables to create one record out of two.

The examples we have used here with such a limited data set would actually yield the same results no matter what kind of `JOIN` was used. Let's move on now and look at the specifics of the different `JOIN` types.

INNER JOINs

`INNER JOIN`s are far and away the most common kind of `JOIN`. They match records together based on one or more common fields, as do most `JOIN`s, but an `INNER JOIN` returns only the records where there are matches for whatever field(s) you have said are to be used for the `JOIN`. In our previous examples, every record was included in the result set at least once, but this situation is rarely the case in the real world.

Let's modify our tables to use an `INNER JOIN`; here's our `Films` table:

FilmID	FilmName	YearMade
1	My Fair Lady	1964
2	Unforgiven	1992

And our `Actors` table:

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn
2	Clint	Eastwood
5	Humphrey	Bogart

Using an `INNER JOIN`, the result set would look like this:

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood

Notice that Bogey was left out of this result set. That's because he didn't have a matching record in the `Films` table. If there isn't a match in both tables, then the record isn't returned. Enough theory — let's try this out in code.

The preferred code for an `INNER JOIN` looks something like this:

```
SELECT <select list>
FROM <first_table>
<join_type> <second_table>
        [ON <join_condition>]
```

This is the ANSI syntax, and you'll have much better luck with it on non-SQL Server database systems than you will if you use the proprietary syntax required prior to version 6.5 (and still used by many developers today). We'll take a look at the other syntax later in the chapter.

> *It is probably worth noting that the term "ANSI syntax" is there because the original foundations of it were created as part of an ANSI standard in the mid 1980s. That standard has since been taken over by the International Standards Organization (ISO), so you may hear it referred to based on either standards organization.*

Fire up the Management Studio and take a test drive of `INNER JOIN`s using the following code against AdventureWorks2008:

```
SELECT *
FROM Person.Person
INNER JOIN HumanResources.Employee
    ON Person.Person.BusinessEntityID = HumanResources.Employee.BusinessEntityID
```

The results of this query are too wide to print in this book, but if you run this, you should get something on the order of 290 rows back. There are several things worth noting about the results:

- The `BusinessEntityID` column appears twice, but there's nothing to say which one is from which table.
- All columns were returned from both tables.
- The first columns listed were from the first table listed.

We can figure out which BusinessEntityID is which just by looking at what table we selected first and matching it with the first BusinessEntityID column that shows up, but this is tedious at best, and at worst, prone to errors. That's one of many reasons why using the plain * operator in JOINs is ill-advised. In the case of an INNER JOIN, however, it's not really that much of a problem because we know that both BusinessEntityID columns, even though they came from different tables, will be exact duplicates of each other. How do we know that? Think about it — since we're doing an INNER JOIN on those two columns, they have to match or the record wouldn't have been returned! Don't get in the habit of counting on this, however. When we look at other JOIN types, we'll find that we can't depend on the JOIN values being equal.

As for all columns being returned from both tables, that is as expected. We used the * operator, which as we've learned before is going to return all columns to us. As I mentioned earlier, the use of the * operator in joins is a bad habit. It's quick and easy, but it's also dirty — it is error-prone and can result in poor performance.

> *As I indicated back in Chapter 3, one good principle to adopt early on is to select what you need and need what you select. What I'm getting at here is that every additional record or column that you return takes up additional network bandwidth and often additional query processing on your SQL Server. The upshot is that selecting unnecessary information hurts performance not only for the current user, but also for every other user of the system and for users of the network on which the SQL Server resides.*
>
> *Select only the columns that you are going to be using and make your* WHERE *clause as restrictive as possible.*

If you insist on using the * operator, you should use it only for the tables from which you need all the columns. That's right — the * operator can be used on a per-table basis. For example, if we wanted all of the base information for our contact, but only needed the Employee table to figure out their JobTitle, we could have changed your query to read:

```
SELECT Person.BusinessEntity.*, JobTitle
FROM Person.BusinessEntity
INNER JOIN HumanResources.Employee
ON Person.BusinessEntity.BusinessEntityID =
HumanResources.Employee.BusinessEntityID
```

If you scroll over to the right in the results of this query, you'll see that most of the Employee-related information is now gone. Indeed, we also only have one instance of the BusinessEntityID column. What we get in our result set contains all the columns from the BusinessEntity table (since we used the * qualified for just that table — your one instance of BusinessEntityID came from this part of the SELECT list) and the only column that had the name JobTitle (which happened to be from the Employee table). Now let's try it again, with only one slight change:

```
SELECT Person.BusinessEntity.*, BusinessEntityID
FROM Person.BusinessEntity
INNER JOIN HumanResources.Employee
        ON Person.BusinessEntity.BusinessEntityID =
HumanResources.Employee.BusinessEntityID
```

Uh, oh — this is a problem. You get an error back:

```
Msg 209, Level 16, State 1, Line 1
Ambiguous column name 'BusinessEntityID'.
```

Why did `JobTitle` work and `BusinessEntityID` not work? For just the reason SQL Server has indicated — our column name is ambiguous. While `JobTitle` exists only in the `Employee` table, `BusinessEntityID` appears in both tables. SQL Server has no way of knowing which one we want. All the instances where we have returned `BusinessEntityID` up to this point have been resolvable: that is, SQL Server could figure out which column was which. In the first query (where we used a plain `*` operator), we asked SQL Server to return everything — that would include *both* `BusinessEntityID` columns, so no name resolution was necessary. In our second example (where we qualified the `*` to be only for `BusinessEntity`), we again said nothing specifically about which `BusinessEntityID` column to use — instead, we said pull everything from the `Contact` table and `BusinessEntityID` just happened to be in that list. `JobTitle` was resolvable because there was only one `JobTitle` column, so that was the one we wanted.

When we want to refer to a column where the column name exists more than once in our `JOIN` result, we must *fully qualify* the column name. We can do this in one of two ways:

- Provide the name of the table that the desired column is from, followed by a period and the column name (`Table.`*`ColumnName`*)
- Alias the tables, and provide that alias, followed by a period and the column name (*`Alias.ColumnName`*), as shown in the previous example

The task of providing the names is straightforward enough — we've already seen how that works with the qualified `*` operator, but let's try our `BusinessEntityID` query again with a qualified column name:

```
SELECT Person. BusinessEntity.*, HumanResources.Employee. BusinessEntityID
FROM Person. BusinessEntity
INNER JOIN HumanResources.Employee
    ON Person. BusinessEntity. BusinessEntityID
        = HumanResources.Employee. BusinessEntityID
```

Now things are working again and the `BusinessEntityID` from the `Employee` table is added to the far right-hand side of the result set.

Aliasing the table is only slightly trickier, but can cut down on the wordiness and help the readability of your query. It works almost exactly the same as aliasing a column in the simple `SELECT`s that we did in the previous chapter — right after the name of the table, we simply state the alias we want to use to refer to that table. Note that, just as with column aliasing, we can use the `AS` keyword (but for some strange reason, this hasn't caught on as much in practice):

```
SELECT pbe.*, hre.BusinessEntityID
FROM Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
    ON pbe.BusinessEntityID = hre.BusinessEntityID
```

Run this code and you'll see that we receive the exact same results as in the previous query.

Be aware that using an alias is an all-or-nothing proposition. Once you decide to alias a table, you must use that alias in every part of the query. This is on a table-by-table basis, but try running some mixed code and you'll see what I mean:

```
SELECT pbe.*, HumanResources.Employee.BusinessEntityID
FROM Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
    ON pbe.BusinessEntityID = hre.BusinessEntityID
```

This seems like it should run fine, but it will give you an error:

```
Msg 4104, Level 16, State 1, Line 1
The multi-part identifier "HumanResources.Employee.BusinessEntityID" could not be
bound.
```

Again, you can mix and match which tables you choose to use aliasing on and which you don't, but once you make a decision for a given table, you have to be consistent in how you reference that table.

Think back to those bullet points we saw a few pages earlier; the columns from the first table listed in the `JOIN` were the first columns returned. Take a break for a moment and think about why that is, and what you might be able to do to control it.

SQL Server always uses a column order that is the best guess it can make at how you want the columns returned. In our first query we used one global `*` operator, so SQL Server didn't have much to go on. In that case, it goes on the small amount that it does have — the order of the columns as they exist physically in the table and the order of tables that you specified in your query. The nice thing is that it is extremely easy to reorder the columns — we just have to be explicit about it. The simplest way to reorder the columns would be to change which table is mentioned first, but we can actually mix and match your column order by simply explicitly stating the columns that we want (even if it is every column), and the order in which we want them.

Try It Out — A Simple JOIN

Let's try a small query to demonstrate the point:

```
SELECT pbe.BusinessEntityID, hre.JobTitle, pp.FirstName, pp.LastName
FROM Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
    ON pbe.BusinessEntityID = hre.BusinessEntityID
INNER JOIN Person.Person pp
 ON pbe.BusinessEntityID = pp.BusinessEntityID
WHERE hre.BusinessEntityID < 4
```

This yields a pretty simple result set:

```
BusinessEntityID  JobTitle                          FirstName      LastName
----------------- ----------------------------------------------------------------
1                 Chief Executive Officer           Ken            Sánchez
2                 Vice President of Engineering     Terri          Duffy
3                 Engineering Manager               Roberto        Tamburello

(3 row(s) affected)
```

How It Works

Unlike when we were nonspecific about what columns we wanted (when we just used the *), this time we were specific about what we wanted. Thus SQL Server knew exactly what to give us — the columns have come out in exactly the order that we've specified in our SELECT list. Indeed, even adding in an additional table, we were able to mix columns between all tables in the order desired.

How an INNER JOIN Is Like a WHERE Clause

In the INNER JOINs that we've done so far, we've really been looking at the concepts that will work for any JOIN type — the column ordering and aliasing is exactly the same for any JOIN. The part that makes an INNER JOIN different from other JOINs is that it is an *exclusive* JOIN — that is, it excludes all records that don't have a value in both tables (the first named, or left table, and the second named, or right table).

Our first example of this was shown in our imaginary Films and Actors tables. Bogey was left out because he didn't have a matching movie in the Films table. Let's look at a real example or two to show how this works.

While you probably haven't realized it in the previous examples, we've already been working with the exclusionary nature of the INNER JOIN. You see, the BusinessEntity table has many, many more rows than the 290 or so that we've been working with. Indeed, our BusinessEntity table has information on virtually any individual our company works with. They can be individuals associated with a particular customer or vendor, or they can be employees. Let's check this out by seeing how many rows exist in the BusinessEntity table:

```
SELECT COUNT(*)
FROM Person.BusinessEntity
```

Run this and, assuming you haven't added any new ones or deleted some, you should come up with approximately 20,777 rows; that's a lot more rows than the 290 that our Employee-related queries have been showing us!

So where did those other 20,487 rows go? As expected, they were excluded from the Employee query result set because there were no corresponding records in the Employee table. It is for this reason that an INNER JOIN is comparable to a WHERE clause. Just as the WHERE clause limits the rows returned to those that match the criteria specified, an INNER JOIN excludes rows because they have no corresponding match in the other table.

Just for a little more proof and practice, let's say we've been asked to produce a list of names associated with at least one customer and the account number of the customers they are associated with. Consider the following tables:

Person.Person	Sales.Customer
BusinessEntityID	CustomerID
PersonType	PersonID
NameStyle	StoreID
Title	TerritoryID
FirstName	AccountNumber
MiddleName	rowguid
LastName	ModifiedDate
Suffix	
EmailPromotion	
AdditionalContactInfo	
Demographics	
rowguid	
ModifiedDate	

Try coming up with this query on your own for a few minutes, then we'll dissect it a piece at a time.

The first thing to do is to figure out what data we need to return. The question calls for two different pieces of information to be returned: the person's name and the account number(s) of the customer they are associated with. The contact's name is available (in parts) from the `Person.Person` table. The customer's account number is available in the `Sales.Customer` table, so we can write the first part of our `SELECT` statement. For brevity's sake, we'll just worry about the first and last name of the contact:

```
SELECT LastName + ', ' + FirstName AS Name, AccountNumber
```

As in many development languages, the + operator can be used for concatenation of strings as well as the addition of numbers. In this case, we are just connecting the last name to the first name with a comma separator in between.

What we need now is something to join the two tables on, and that's where we run into our first problem — there doesn't appear to be one. The tables don't seem to have anything in common on which we can base our `JOIN` – fortunately, looks can be deceiving.

If we were to look more closely at the definition of the Sales.Customer table, we would find that the `Customer.PersonID` column has a *foreign key* (an indicator that a given column is dependent on information from another column). Indeed, the `PersonID` ties back to the `BusinessEntityID` in the `Person.Person` table.

This is probably not a bad time to point out how I'm not really a fan of the AdventureWorks2008 database. I'll even go so far as to apologize for using it, but my publisher essentially did the writer's equivalent of holding me at gun point and forced the migration to AdventureWorks2008 as the sample for this book. The naming of these columns is one of many examples where the AdventureWorks2008 structure is almost bizarre, but we work with what we have.

The good news is that there is no requirement that the columns that we join share the same name, so we can just join the columns specifying the appropriate names for each table.

Try It Out More Complex JOINs

Using this mismatched pair of names in your `JOIN` is no problem — we just keep on going with our `FROM` clause and `JOIN` keywords (don't forget to switch the database to pubs):

```
SELECT CAST(LastName + ', ' + FirstName AS varchar(35)) AS Name, AccountNumber
FROM Person.Person pp
JOIN Sales.Customer sc
 ON pp.BusinessEntityID = sc.PersonID
```

Our `SELECT` statement is now complete! If we execute, we get something like:

```
Name                        AccountNumber
--------------------------- -------------
Robinett, David             AW00011377
Robinson, Rebecca           AW00011913
Robinson, Dorothy           AW00011952
Rockne, Carol Ann           AW00020164
Rodgers, Scott              AW00020211
Rodman, Jim                 AW00020562
…
…
He, Crystal                 AW00024634
Zheng, Crystal              AW00021127
Hu, Crystal                 AW00027980

(19119 row(s) affected)
```

Note that your sort order and, therefore, the actual names you see at the top and bottom of your results may differ from what you see here. Remember, SQL Server makes no promises about the order your results will arrive in unless you use an `ORDER BY` *clause — since we didn't use* `ORDER BY`*, the old adage "actual results may vary" comes into play.*

How It Works

If we were to do a simple `SELECT *` against the `Person` table, we would find that several contacts would be left out because, although they are considered people, they apparently aren't customers (they may be employees, vendor contacts, or perhaps just potential customers). Once again, the key to `INNER JOIN`s is that they are exclusive.

Notice that we did not use the `INNER` *keyword in the query. That is because an* `INNER JOIN` *is the default* `JOIN` *type. Schools of thought vary on this, but I believe that because leaving the* `INNER` *keyword out has dominated the way code has been written for so long, it is almost more confusing to put it in — that's why you won't see me use it again in this book.*

OUTER JOINs

This type of `JOIN` is something of the exception rather than the rule. This is definitely not because they don't have their uses, but rather because:

- We, more often than not, want the kind of exclusiveness that an inner join provides.
- Many SQL writers learn inner joins and never go any further — they simply don't understand the outer variety.
- There are often other ways to accomplish the same thing.
- They are often simply forgotten about as an option.

Whereas inner joins are exclusive in nature, outer and, as we'll see later in this chapter, full joins are inclusive. It's a tragedy that people don't get to know how to make use of outer joins because they make seemingly difficult questions simple. They can also often speed performance when used instead of nested subqueries (which we will look into in Chapter 7).

Earlier in this chapter, we introduced the concept of a join having sides — a left and a right. The first named table is considered to be on the left and the second named table is considered to be on the right. With inner joins these are a passing thought at most, because both sides are always treated equally. With outer joins, however, understanding your left from your right is absolutely critical. When you look at it, it seems very simple because it is very simple, yet many query mistakes involving outer joins stem from not thinking through your left from your right.

To learn how to construct outer joins correctly, we're going to use two syntax illustrations. The first deals with the simple scenario of a two-table outer join. The second will deal with the more complex scenario of mixing outer joins with any other join.

The Simple OUTER JOIN

The first syntax situation is the easy part — most people get this part just fine:

```
SELECT <SELECT list>
FROM <the table you want to be the "LEFT" table>
<LEFT|RIGHT> [OUTER] JOIN <table you want to be the "RIGHT" table>
                       ON <join condition>
```

In the examples, you'll find that I tend to use the full syntax — that is, I include the `OUTER` *keyword (for example,* `LEFT OUTER JOIN`*). Note that the* `OUTER` *keyword is optional — you need only include the* `LEFT` *or* `RIGHT` *(for example,* `LEFT JOIN`*). In practice, I find that the* `OUTER` *keyword is rarely used.*

What I'm trying to get across here is that the table that comes before the `JOIN` keyword is considered to be the `LEFT` table, and the table that comes after the `JOIN` keyword is considered to be the `RIGHT` table.

`OUTER JOIN`s are, as I've said, inclusive in nature. What specifically gets included depends on which side of the join you have emphasized. A `LEFT OUTER JOIN` includes all the information from the table on the left, and a `RIGHT OUTER JOIN` includes all the information from the table on the right. Let's put this into practice with a small query so that you can see what I mean.

Let's say we want to know what all our special offers are, the amount of each discount, and which products, if any, can have them applied. Looking over the AdventureWorks2008 database, we have a table called `SpecialOffer` in the Sales schema. We also have an associate table called `SpecialOfferProduct` that lets us know what special offers are associated with what products:

Sales.SpecialOffer	**Sales.SpecialOfferProduct**
SpecialOfferID	SpecialOfferID
Description	ProductID
DiscountPct	rowguid
Type	ModifiedDate
Category	
StartDate	
EndDate	
MinQty	
MaxQty	
rowguid	
ModifiedDate	

We can directly join these tables based on the `SpecialOfferID`. If we did this using a common `INNER JOIN`, it would look something like:

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso
JOIN Sales.SpecialOfferProduct ssop
ON sso.SpecialOfferID = ssop.SpecialOfferID
WHERE sso.SpecialOfferID != 1
```

Note that I'm deliberately eliminating the rows with no discount (that's `SpecialOfferID 1`). This query yields 243 rows — each with an associated `ProductID`:

```
SpecialOfferID Description                  DiscountPct    ProductID
-------------- ------------------------------ -------------- -----------
2              Volume Discount 11 to 14       0.02           707
2              Volume Discount 11 to 14       0.02           708
2              Volume Discount 11 to 14       0.02           709
2              Volume Discount 11 to 14       0.02           711
…
…

…
…
16             Mountain-500 Silver Clearance  0.40           986
16             Mountain-500 Silver Clearance  0.40           987
16             Mountain-500 Silver Clearance  0.40           988

(243 row(s) affected)
```

Think about this, though. We wanted results based on the special offers we have — not which ones were actually in use. This query only gives us special offers that have products utilizing the offer — it doesn't answer the question!

What we need is something that's going to return every special offer and the product ids where applicable.

Try It Out — Outer JOINs

In order to return every special offer and the products where applicable, we need to change only the `JOIN` type in the query:

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso
LEFT OUTER JOIN Sales.SpecialOfferProduct ssop
 ON sso.SpecialOfferID = ssop.SpecialOfferID
WHERE sso.SpecialOfferID != 1
```

This yields similar results, but with one rather important difference:

```
SpecialOfferID Description                  DiscountPct    ProductID
-------------- ------------------------------ -------------- -----------
2              Volume Discount 11 to 14       0.02           707
2              Volume Discount 11 to 14       0.02           708
2              Volume Discount 11 to 14       0.02           709
2              Volume Discount 11 to 14       0.02           711
…
…
6              Volume Discount over 60        0.20           NULL
…
…
16             Mountain-500 Silver Clearance  0.40           986
16             Mountain-500 Silver Clearance  0.40           987
```

```
16            Mountain-500 Silver Clearance  0.40          988

(244 row(s) affected)
```

If you were to perform a `SELECT *` against the `discounts` table, you'd quickly find that we have included every row from that table except for `SpecialOfferID 1`, which we explicitly excluded from our results.

How It Works

We are doing a `LEFT JOIN`, and the `SpecialOffer` table is on the left side of the `JOIN`. But what about the `SpecialOfferProduct` table? If we are joining and we don't have a matching record for the `SpecialOfferProduct` table, then what happens? Since it is not on the inclusive side of the `JOIN` (in this case, the `LEFT` side), SQL Server will fill in a `NULL` for any value that comes from the opposite side of the join if there is no match with the inclusive side of the `JOIN`. In this case, all but one of our rows have `ProductID`s. What we can discern from that is that all of our `SpecialOffers` are associated with at least one product except one (`SpecialOfferID 6`).

We've answered the question then; of the 16 `SpecialOffers` available, only one is not being used (Volume Discount over 60).

In this case, switching to a `RIGHT JOIN` would yield the same thing as the `INNER JOIN`, as the `SpecialOfferProduct` table only contains rows where there is an active link between a special offer and a product. The concept is, however, exactly the same. We could, for example, switch the order that we reference the tables and then use a `RIGHT JOIN`.

Try It Out — RIGHT OUTER JOINs

Now, let's see what happens if we change the question to a `RIGHT OUTER JOIN`:

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOfferProduct ssop
RIGHT OUTER JOIN Sales.SpecialOffer sso
 ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE sso.SpecialOfferID != 1
```

How It Works

If we tried the preceding query with a `LEFT JOIN`, we would get back just the 243 rows we got with the `INNER JOIN`. Run it as presented above, and we get back the unused special offer we wanted.

Finding Orphan or Non-Matching Records

We can actually use the inclusive nature of `OUTER JOIN`s to find non-matching records in the exclusive table. What do I mean by that? Let's look at an example.

Let's change our special offer question. We want to know the store names for all the special offers that are not associated with any products. Can you come up with a query to perform this based on what we

know thus far? Actually, the very last query we ran has us 90 percent of the way there. Think about it for a minute: an OUTER JOIN returns a NULL value in the ProductID column wherever there is no match. What we are looking for is pretty much the same result set we received in the previous query, except that we want to filter out any records that do have a ProductID, and we want only the special offer name. To do this, we simply change our SELECT list and add an extra condition to the WHERE clause:

```
SELECT Description
FROM Sales.SpecialOfferProduct ssop
RIGHT OUTER JOIN Sales.SpecialOffer sso
 ON ssop.SpecialOfferID = sso.SpecialOfferID
WHERE sso.SpecialOfferID != 1
  AND ssop.SpecialOfferID IS NULL
```

As expected, we have exactly the same stores that had NULL values before:

```
Description
------------------------------
Volume Discount over 60

(1 row(s) affected)
```

There is one question you might be thinking at the moment that I want to answer in anticipation, so that you're sure you understand why this will always work. The question is: "What if the discount record really has a NULL value?" Well, that's why we built a WHERE clause on the same field that was part of our JOIN. If we are joining based on the stor_id columns in both tables, then only three conditions can exist:

- If the SpecialOfferProduct.SpecialOfferID column has a non-NULL value, then, according to the ON operator of the JOIN clause, if a special offer record exists, then SpecialOffer.SpecialOfferID must also have the same value as SpecialOfferProduct.SpecialOfferID (look at the ON ssop.SpecialOfferID = sso.SpecialOfferID).
- If the SpecialOfferProduct.SpecialOfferID column has a non-NULL value, then, according to the ON operator of the JOIN clause, if a special offer record does not exist, then SpecialOffer.SpecialOfferID will be returned as NULL.
- If the SpecialOfferProduct.SpecialOfferID happens to have a NULL value, and SpecialOffer.SpecialOfferID also has a NULL value, there will be no join (null does not equal null), and SpecialOffer.SpecialOfferID will return NULL because there is no matching record.

A value of NULL does not join to a value of NULL. Why? Think about what we've already said about comparing NULLs — a NULL does not equal NULL. Be extra careful of this when coding. One of the more common questions I am asked is, "Why isn't this working?" in a situation where people are using an "equal to" operation on a NULL — it simply doesn't work because they are not equal. If you want to test this, try executing some simple code:

```
IF (NULL=NULL)
   PRINT 'It Does'
ELSE
   PRINT 'It Doesn''t'
```

If you execute this, you'll get the answer to whether your SQL Server thinks a NULL equals a NULL — that is, it doesn't.

Let's use this notion of being able to identify non-matching records to locate some of the missing records from one of our earlier `INNER JOIN`s. Remember these two queries, which you ran against AdventureWorks2008?

```
SELECT pbe.BusinessEntityID, hre.JobTitle, pp.FirstName, pp.LastName
FROM Person.BusinessEntity pbe
INNER JOIN HumanResources.Employee hre
    ON pbe.BusinessEntityID = hre.BusinessEntityID
INNER JOIN Person.Person pp
 ON pbe.BusinessEntityID = pp.BusinessEntityID
WHERE hre.BusinessEntityID < 4
```

And . . .

```
SELECT COUNT(*)
FROM Person.BusinessEntity
```

The first was one of our queries where we explored the `INNER JOIN`. We discovered by running the second query that the first had excluded (by design) some rows. Now let's identify the excluded rows by using an `OUTER JOIN`.

We know from our `SELECT COUNT(*)` query that our first query is missing thousands of records from the `BusinessEntity` table. (It could conceivably be missing records from the `Employee` table, but we're not interested in that at the moment.) The implication is that there are records in the `BusinessEntity` table that do not have corresponding `Employee` records. This makes sense, of course, because we know that some of our persons are customers or vendor contacts. While our manager's first question was about all the employee contact information, it would be very common to ask just the opposite: "Which persons are not employees?" That question is answered with the same result obtained by asking, "Which records exist in `Person` that don't have corresponding records in the `Employee` table?" The solution has the same structure as the query to find special offers that aren't associated with any products:

```
SELECT  pp.BusinessEntityID, pp.FirstName, pp.LastName
FROM Person.Person pp
LEFT OUTER JOIN HumanResources.Employee hre
 ON pp.BusinessEntityID = hre.BusinessEntityID
WHERE hre.BusinessEntityID IS NULL
```

Just that quickly we have a list of contacts that is somewhat cleaned up to contain just customers:

```
BusinessEntityID FirstName             LastName
---------------- --------------------- --------------------------------
293              Catherine             Abel
295              Kim                   Abercrombie
2170             Kim                   Abercrombie
…
…
2088             Judy                  Zugelder
12079            Jake                  Zukowski
2089             Michael               Zwilling

(19682 row(s) affected)
```

Note that whether you use a `LEFT` or a `RIGHT JOIN` doesn't matter as long as the correct table or group of tables is on the corresponding side of the `JOIN`. For example, we could have run the preceding query using a `RIGHT JOIN` as long as we also switched which sides of the `JOIN` the `Person` and `Employee` tables were on. For example, this would have yielded exactly the same results:

```
SELECT  pp.BusinessEntityID, pp.FirstName, pp.LastName
FROM HumanResources.Employee hre
RIGHT OUTER JOIN Person.Person pp
 ON pp.BusinessEntityID = hre.BusinessEntityID
WHERE hre.BusinessEntityID IS NULL
```

When we take a look at even more advanced queries, we'll run into a slightly more popular way of finding records that exist in one table without there being corresponding records in another table. Allow me to preface that by saying that using `JOIN`s is usually our best bet in terms of performance. There are exceptions to the rule that we will cover as we come across them, but in general, the use of `JOIN`s will be best when faced with multiple options.

Dealing with More Complex OUTER JOINs

Now we're on to our second illustration and how to make use of it. This scenario is all about dealing with an `OUTER JOIN` mixed with some other `JOIN` (no matter what the variety).

It is when combining an `OUTER JOIN` with other `JOIN`s that the concept of sides becomes even more critical. What's important to understand here is that everything to the "left" — or before — the `JOIN` in question will be treated just as if it were a single table for the purposes of inclusion or exclusion from the query. The same is true for everything to the "right" — or after — the `JOIN`. The frequent mistake here is to perform a `LEFT OUTER JOIN` early in the query and then use an `INNER JOIN` late in the query. The `OUTER JOIN` includes everything up to that point in the query, but the `INNER JOIN` may still create a situation where something is excluded! My guess is that you will, like most people (including me for a while), find this exceptionally confusing at first, so let's see what we mean with some examples. Because none of the databases that come along with SQL Server has any good scenarios for demonstrating this, we're going to have to create a database and sample data of our own.

If you want to follow along with the examples, the example database called Chapter4DB can be created by running `Chapter4DB.sql` from the downloadable source code. Simply open the file in the Management Studio query window and execute it.

Again, in order to utilize the next several examples, you must execute the `Chapter4DB.sql` script included in the downloadable code for this book.

What we are going to do is to build up a query step-by-step and watch what happens. The query we are looking for will return a vendor name and the address of that vendor. The example database only has a few records in it, so let's start out by selecting all the choices from the central item of the query — the

vendor. We're going to go ahead and start aliasing from the beginning, since we will want to do this in the end:

```
USE Chapter4DB

SELECT v.VendorName
FROM Vendors v
```

This yields a scant three records:

```
VendorName
--------------------------------------
Don's Database Design Shop
Dave's Data
The SQL Sequel

(3 row(s) affected)
```

These are the names of every vendor that we have at this time. Now let's add in the address information — there are two issues here. First, we want the query to return every vendor no matter what, so we'll make use of an `OUTER JOIN`. Next, a vendor can have more than one address and vice versa, so the database design has made use of an associate table. This means that we don't have anything to directly join the `Vendors` and `Address` tables — we must instead join both of these tables to our linking table, which is called `VendorAddress`. Let's start out with the logical first piece of this join:

```
SELECT v.VendorName
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
             ON v.VendorID = va.VendorID
```

Because `VendorAddress` doesn't itself have the address information, we're not including any columns from that table in our `SELECT` list. `VendorAddress`'s sole purpose in life is to be the connection point of a many-to-many relationship (one vendor can have many addresses and, as we've set it up here, an address can be the home of more than one vendor). Running this, as we expect, gives us the same results as before:

```
VendorName
--------------------------------------
Don's Database Design Shop
Dave's Data
The SQL Sequel

(3 row(s) affected)
```

Let's take a brief time-out from this particular query to check on the table against which we just joined. Try selecting all the data from the `VendorAddress` table:

```
SELECT *
FROM VendorAddress
```

Just two records are returned:

```
VendorID     AddressID
--------     -------------
1            1
2            3

(2 row(s) affected)
```

We know, therefore, that our `OUTER JOIN` is working. Since there are only two records in the `VendorAddress` table and three vendors are returned, we must be returning at least one row from the `Vendors` table that didn't have a matching record in the `VendorAddress` table. While we're here, we'll just verify that by briefly adding one more column back to our vendors query:

```
SELECT v.VendorName, va.VendorID
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
             ON v.VendorID = va.VendorID
```

Sure enough, we wind up with a `NULL` in the `VendorID` column from the `VendorAddress` table:

```
VendorName                   VendorID
--------------------         --------------
Don's Database Design Shop   1
Dave's Data                  2
The SQL Sequel               NULL

(3 row(s) affected)
```

The vendor named "The SQL Sequel" would not have been returned if we were using an `INNER` or `RIGHT JOIN`. Our use of a `LEFT JOIN` has ensured that we get all vendors in our query result.

Now that we've tested things out a bit, let's return to our original query and then add in the second `JOIN` to get the actual address information. Because we don't care if we get all addresses, no special `JOIN` is required — at least, it doesn't appear that way at first . . .

```
SELECT v.VendorName, a.Address
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
             ON v.VendorID = va.VendorID
JOIN Address a
   ON va.AddressID = a.AddressID
```

We get back the address information as expected, but there's a problem:

```
VendorName                    Address
---------------------------   ---------------
Don's Database Design Shop    1234 Anywhere
Dave's Data                   567 Main St.

(2 row(s) affected)
```

Somehow, we lost one of our vendors. That's because SQL Server is applying the rules in the order that we've stated them. We have started with an `OUTER JOIN` between `Vendors` and `VendorAddress`. SQL Server does just what we want for that part of the query — it returns all vendors. The issue comes when it applies the next set of instructions. We have a result set that includes all the vendors, but we now apply that result set as part of an `INNER JOIN`. Because an `INNER JOIN` is exclusive to both sides of the `JOIN`, only records where the result of the first `JOIN` has a match with the second `JOIN` will be included. Because only two records match up with a record in the `Address` table, only two records are returned in the final result set. We have two ways of addressing this:

- Add yet another `OUTER JOIN`
- Change the order of the `JOIN`s

Let's try it both ways. We'll add another `OUTER JOIN` first:

```
SELECT v.VendorName, a.Address
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
             ON v.VendorID = va.VendorID
LEFT OUTER JOIN Address a
             ON va.AddressID = a.AddressID
```

And now we get our expected result:

```
VendorName                     Address
-------------------------      ------------
Don's Database Design Shop     1234 Anywhere
Dave's Data                    567 Main St.
The SQL Sequel                 NULL

(3 row(s) affected)
```

Now do something slightly more dramatic and reorder our original query:

```
SELECT v.VendorName, a.Address
FROM VendorAddress va
JOIN Address a
   ON va.AddressID = a.AddressID
RIGHT OUTER JOIN Vendors v
              ON v.VendorID = va.VendorID
```

And we still get our desired result:

```
VendorName                    Address
-------------------------     -------------
Don's Database Design Shop    1234 Anywhere
Dave's Data                   567 Main St.
The SQL Sequel                NULL

(3 row(s) affected)
```

The question you should be asking now is, "Which way is best?" Quite often in SQL, there are several ways of executing the query without one having any significant advantage over the other — this is *not* one of those times.

I would most definitely steer you to the second of the two solutions.

> **The rule of thumb is to get all of the `INNER JOIN`s you can out of the way first; you will then find yourself using the minimum number of `OUTER JOIN`s and decreasing the number of errors in your data.**

The reason has to do with navigating as quickly as possible to your data. If you keep adding `OUTER JOIN`s not because of what's happening with the current table you're trying to add in, but because you're trying to carry through an earlier `JOIN` result, you are much more likely to include something you don't intend, or to make some sort of mistake in your overall logic. The second solution addresses this by using only the `OUTER JOIN` where necessary — just once. You can't always create a situation where the `JOIN`s can be moved around to this extent, but you often can.

> *I can't stress enough how often I see errors with `JOIN` order. It is one of those areas that just seems to give developers fits. Time after time I get called in to look over a query that someone has spent hours verifying each section of, and it seems that at least half the time I get asked whether I know about this SQL Server "bug." The bug isn't in SQL Server — it's with the developer. If you take anything away from this section, I hope it is that `JOIN` order is one of the first places to look for errors when the results aren't coming up as you expect.*

Seeing Both Sides with FULL JOINs

Like many things in SQL, a `FULL JOIN` (also known as a `FULL OUTER JOIN`) is basically what it sounds like — it is a matching up of data on both sides of the `JOIN` with everything included, no matter what side of the `JOIN` it is on.

`FULL JOIN`s seem really cool when you learn them and then almost never get used. You'll find an honest politician more often than you'll find a `FULL JOIN` in use. Their main purpose in life is to look at the complete relationship between data without giving preference to one side or the other. You want to know about every record on both sides of the equation — with nothing left out.

A `FULL JOIN` is perhaps best described as what you would get if you could do a `LEFT JOIN` and a `RIGHT JOIN` in the same `JOIN`. You get all the records that match, based on the `JOIN` field(s). You also get any records that exist only on the left side, with `NULL`s being returned for columns from the right side. Finally, you get any records that exist only on the right side, with `NULL`s being returned for columns from the left side. Note that, when I say finally, I don't mean to imply that they'll be last in the query. The result order you get will (unless you use an `ORDER BY` clause) depend entirely on what SQL Server thinks is the least costly way to retrieve your records.

Try It Out FULL JOINs

Let's just get right to it by looking back at our previous query from the section on `OUTER JOIN`s:

```
SELECT v.VendorName, a.Address
FROM VendorAddress va
JOIN Address a
   ON va.AddressID = a.AddressID
RIGHT OUTER JOIN Vendors v
              ON v.VendorID = va.VendorID
```

What we want to do here is take it a piece at a time again and add some fields to the `SELECT` list that will let us see what's happening. First, we'll take the first two tables using a `FULL JOIN`:

```
SELECT a.Address, va.AddressID
FROM VendorAddress va
FULL JOIN Address a
       ON va.AddressID = a.AddressID
```

As it happens, a `FULL JOIN` on this section doesn't yield any more than a `RIGHT JOIN` would have:

```
Address              AddressID
--------------       -------------
1234 Anywhere        1
567 Main St.         3
999 1st St.          NULL
1212 Smith Ave       NULL
364 Westin           NULL

(5 row(s) affected)
```

But wait — there's more! Now add the second `JOIN`:

```
SELECT a.Address, va.AddressID, v.VendorID, v.VendorName
FROM VendorAddress va
FULL JOIN Address a
       ON va.AddressID = a.AddressID
FULL JOIN Vendors v
       ON va.VendorID = v.VendorID
```

Now we have everything:

```
Address              AddressID      VendorID      VendorName
--------------       ----------     ----------    ------------------------------
1234 Anywhere        1              1             Don's Database Design Shop
567 Main St.         3              2             Dave's Data
999 1st St.          NULL           NULL          NULL
1212 Smith Ave       NULL           NULL          NULL
364 Westin           NULL           NULL          NULL
NULL                 NULL           3             The SQL Sequel

(6 row(s) affected)
```

How It Works

As you can see, we have the same two rows that we would have had with an INNER JOIN clause. Those are then followed by the three Address records that aren't matched with anything in either table. Last, but not least, we have the one record from the Vendors table that wasn't matched with anything.

Again, use a FULL JOIN when you want all records from both sides of the JOIN — matched where possible, but included even if there is no match.

CROSS JOINs

CROSS JOINs are very strange critters indeed. A CROSS JOIN differs from other JOINs in that there is no ON operator and that it joins every record on one side of the JOIN with every record on the other side of the JOIN. In short, you wind up with a Cartesian product of all the records on both sides of the JOIN. The syntax is the same as any other JOIN, except that it uses the keyword CROSS (instead of INNER, OUTER, or FULL) and that it has no ON operator. Here's a quick example:

```
SELECT v.VendorName, a.Address
FROM Vendors v
CROSS JOIN Address a
```

Think back now — we had three records in the Vendors table and five records in the Address table. If we're going to match every record in the Vendors table with every record in the Address table, then we should end up with 3 * 5 = 15 records in our CROSS JOIN:

```
VendorName                     Address
-------------------------      --------------------
Don's Database Design Shop     1234 Anywhere
Don's Database Design Shop     567 Main St.
Don's Database Design Shop     999 1st St.
Don's Database Design Shop     1212 Smith Ave
Don's Database Design Shop     364 Westin
Dave's Data                    1234 Anywhere
Dave's Data                    567 Main St.
Dave's Data                    999 1st St.
Dave's Data                    1212 Smith Ave
Dave's Data                    364 Westin
The SQL Sequel                 1234 Anywhere
The SQL Sequel                 567 Main St.
The SQL Sequel                 999 1st St.
The SQL Sequel                 1212 Smith Ave
The SQL Sequel                 364 Westin

(15 row(s) affected)
```

Indeed, that's exactly what we get.

Every time I teach a SQL class, I get asked the same question about CROSS JOINs: "Why in the world would you use something like this?" I'm told there are scientific uses for it — this makes sense to me since I know there are a number of high-level mathematical functions that make use of Cartesian products. I presume that you could read a large number of samples into table structures, and then perform your CROSS JOIN to create a Cartesian product of your sample. There is, however, a much more frequently occurring use for CROSS JOINs — the creation of test data.

When you are building up a database, that database is quite often part of a larger-scale system that will need substantial testing. A recurring problem in testing of large-scale systems is the creation of large amounts of test data. By using a CROSS JOIN, you can do smaller amounts of data entry to create your test data in two or more tables, and then perform a CROSS JOIN against the tables to produce a much larger set of test data. You have a great example in the previous query — if you needed to match a group of addresses up with a group of vendors, then that simple query yields 15 records from 8. Of course, the numbers can become far more dramatic. For example, if we created a table with 50 first names and then created a table with 250 last names, we could CROSS JOIN them together to create a table with 12,500 unique name combinations. By investing in keying in 300 names, we suddenly get a set of test data with 12,500 names.

Exploring Alternative Syntax for Joins

What we're going to look at in this section is what many people still consider to be the "normal" way of coding joins. Until SQL Server 6.5, the alternative syntax we'll look at here was the only join syntax in SQL Server, and what is today called the "standard" way of coding joins wasn't even an option.

Until now, we have been using the ANSI/ISO syntax for all of our SQL statements. I highly recommend that you use the ANSI method since it has much better portability between systems and is also much more readable. It is worth noting that the old syntax is actually reasonably well supported across platforms at the current time.

> *The primary reason I am covering the old syntax at all is that there is absolutely no doubt that, sooner or later, you will run into it in legacy code. I don't want you staring at that code saying, "What the heck is this?"*
>
> *That being said, I want to reiterate my strong recommendation that you use the ANSI syntax wherever possible. Again, it is substantially more readable and Microsoft has indicated that they may not continue to support the old syntax indefinitely. I find it very hard to believe, given the amount of legacy code out there, that Microsoft will dump the old syntax any time soon, but you never know.*
>
> *Perhaps the biggest reason is that the ANSI syntax is actually more functional. Under old syntax, it was actually possible to create ambiguous query logic — where there was more than one way to interpret the query. The new syntax eliminates this problem.*

Remember when I compared a JOIN to a WHERE clause earlier in this chapter? Well, there was a reason. The old syntax expresses all of the JOINs within the WHERE clause.

The old syntax supports all of the JOINs that we've done using ANSI with the exception of a FULL JOIN. If you need to perform a FULL JOIN, I'm afraid you'll have to stick with the ANSI version.

An Alternative INNER JOIN

Let's do a déjà vu thing and look back at the first INNER JOIN we did in this chapter:

```
USE AdventureWorks2008

SELECT *
FROM Person.Person
INNER JOIN HumanResources.Employee
    ON Person.Person.BusinessEntityID = HumanResources.Employee.BusinessEntityID
```

This got us approximately 290 rows back. Instead of using the JOIN, however, let's rewrite it using a WHERE-clause–based join syntax. It's actually quite easy — just eliminate the words INNER JOIN, add a comma, and replace the ON operator with a WHERE clause:

```
SELECT *
FROM Person.Person, HumanResources.Employee
WHERE Person.Person.BusinessEntityID = HumanResources.Employee.BusinessEntityID
```

It's a piece of cake and it yields us the same 290 rows we got with the other syntax.

This syntax is supported by virtually all major SQL systems (Oracle, DB2, MySQL, and so on) in the world today, but can create some ambiguity at what point in the query processing the restriction should be applied. It is very rare to run into such an ambiguity, but it can happen, so use the JOIN *syntax for any new queries and edit old queries to the new syntax as you are able.*

An Alternative OUTER JOIN

> **Note that the alternative syntax for** OUTER **joins is only available if you tell SQL Server you want to run in SQL Server 2000 compatibility mode (setting the compatibility level to 80 in the** ALTER DATABASE **command).**

The alternative syntax for OUTER JOINs works pretty much the same as the INNER JOIN, except that, because we don't have the LEFT or RIGHT keywords (and no OUTER or JOIN for that matter), we need some special operators especially built for the task. These look like this:

Alternative	ANSI
*=	LEFT JOIN
=*	RIGHT JOIN

Let's pull up the first OUTER JOIN we did in this chapter. It made use of the pubs database and looked something like this:

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso
```

```
JOIN Sales.SpecialOfferProduct ssop
ON sso.SpecialOfferID = ssop.SpecialOfferID
WHERE sso.SpecialOfferID != 1
```

Again, we just lose the words `LEFT OUTER JOIN` and replace the `ON` operator with a `WHERE` clause, or, in this case, add the `ON` condition to the existing `WHERE` clause:

```
SELECT sso.SpecialOfferID, Description, DiscountPct, ProductID
FROM Sales.SpecialOffer sso,
     Sales.SpecialOfferProduct ssop
WHERE sso.SpecialOfferID *= ssop.SpecialOfferID
  AND sso.SpecialOfferID != 1
```

If you were to run this (I'd recommend against doing the change in compatibility level that would be required, but I want you to know that this kind of code is out there), we would get back the same results we did in our original `OUTER JOIN` query. A `RIGHT JOIN` would function in much the same way.

An Alternative CROSS JOIN

This is far and away the easiest of the bunch. To create a `CROSS JOIN` using the old syntax, you just do nothing. That is, you don't put anything in the `WHERE` clause of the `FROM`: `TableA.ColumnA = TableB.ColumnA`.

So, for an ultra quick example, let's take the first example from the `CROSS JOIN` section earlier in the chapter. The ANSI syntax looked like this:

```
USE Chapter4DB

SELECT v.VendorName, a.Address
FROM Vendors v
CROSS JOIN Address a
```

To convert it to the old syntax, we just strip out the `CROSS JOIN` keywords and add a comma:

```
USE Chapter4DB

SELECT v.VendorName, a.Address
FROM Vendors v, Address a
```

As with the other examples in this section, we get back the same results that we got with the ANSI syntax:

```
VendorName                     Address
-------------------------      --------------------
Don's Database Design Shop     1234 Anywhere
Don's Database Design Shop     567 Main St.
Don's Database Design Shop     999 1st St.
Don's Database Design Shop     1212 Smith Ave
Don's Database Design Shop     364 Westin
Dave's Data                    1234 Anywhere
Dave's Data                    567 Main St.
Dave's Data                    999 1st St.
```

```
Dave's Data                    1212 Smith Ave
Dave's Data                    364 Westin
The SQL Sequel                 1234 Anywhere
The SQL Sequel                 567 Main St.
The SQL Sequel                 999 1st St.
The SQL Sequel                 1212 Smith Ave
The SQL Sequel                 364 Westin

(15 row(s) affected)
```

This is supported across all versions and across most of the database management systems.

The UNION

OK, enough with all the "old syntax" versus "new syntax" stuff. Now we're into something that's the same regardless of what other join syntax you prefer — the `UNION` operator. `UNION` is a special operator we can use to cause two or more queries to generate one result set.

A `UNION` isn't really a `JOIN`, like the previous options we've been looking at — instead it's more of an appending of the data from one query right onto the end of another query (functionally, it works a little differently than this, but this is the easiest way to look at the concept). Where a `JOIN` combined information horizontally (adding more columns), a `UNION` combines data vertically (adding more rows), as illustrated in Figure 4-1.

When dealing with queries that use a `UNION`, there are just a few key points:

- All the `UNION`ed queries must have the same number of columns in the `SELECT` list. If your first query has three columns in the `SELECT` list, then the second (and any subsequent queries being `UNION`ed) must also have three columns. If the first has five, then the second must have five, too. Regardless of how many columns are in the first query, there must be the same number in the subsequent query(s).
- The headings returned for the combined result set will be taken only from the first of the queries. If your first query has a `SELECT` list that looks like `SELECT Col1, Col2 AS Second, Col3 FROM...`, then regardless of how your columns are named or aliased in the subsequent queries, the headings on the columns returned from the `UNION` will be `Col1`, `Second`, and `Col3` respectively.
- The data types of each column in a query must be implicitly compatible with the data type in the same relative column in the other queries. Note that I'm *not* saying they have to be the same data type — they just have to be implicitly convertible (a conversion table that shows implicit versus explicit conversions can be found in Figure 1-3 of Chapter 1). If the second column in the first query were of type `char(20)`, then it would be fine if the second column in the second query were `varchar(50)`. However, because things are based on the first query, any rows longer than 20 would be truncated for data from the second result set.
- Unlike non-`UNION` queries, the default return option for `UNION`s is `DISTINCT` rather than `ALL`. This can really be confusing to people. In your other queries, all rows were returned regardless of whether they were duplicated with another row or not, but the results of a `UNION` do not work that way. Unless you use the `ALL` keyword in your query, only one of any repeating rows will be returned.

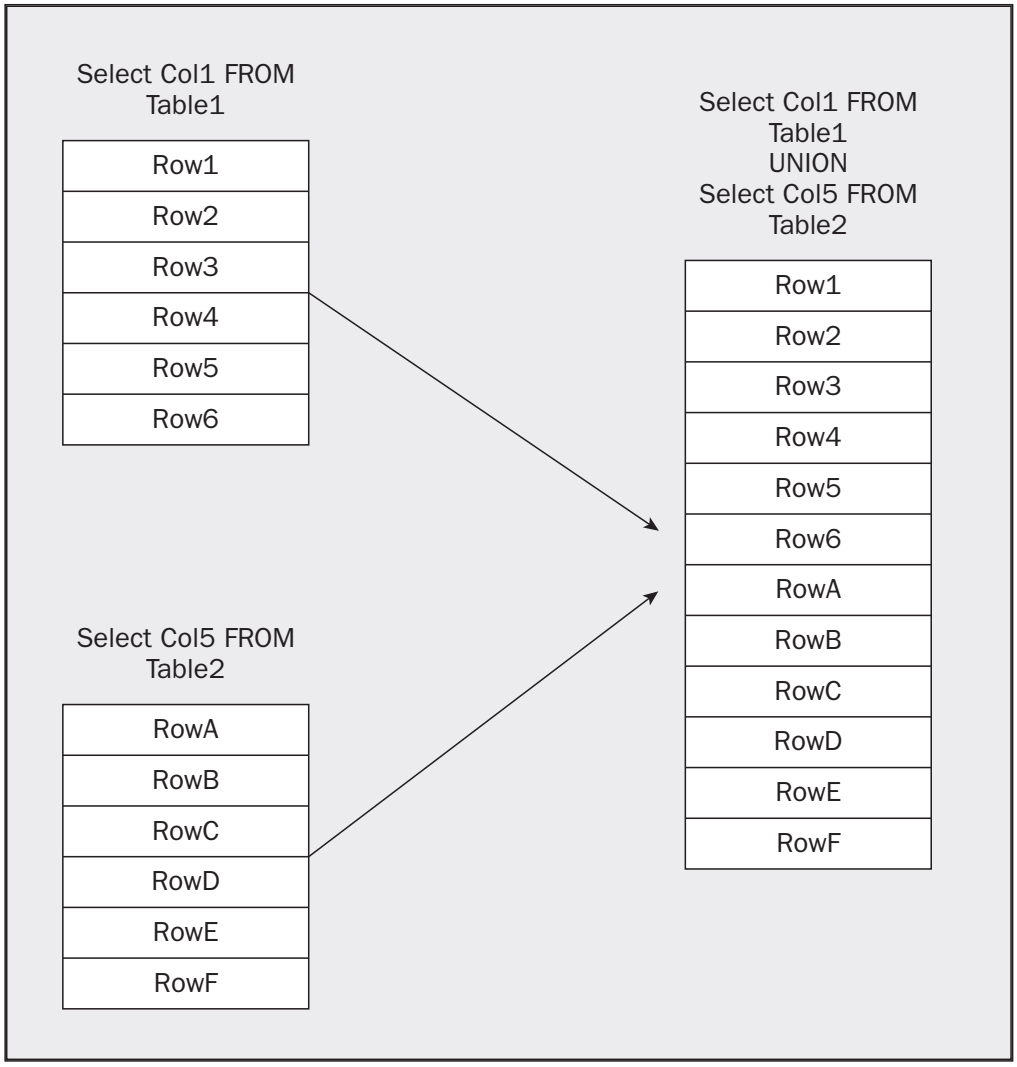

Figure 4-1

As always, let's take a look at this with an example or two.

Try It Out UNION

First, let's look at a `UNION` that has some practical use to it. (It's something I could see happening in the real world — albeit not all that often.) For this example, we're going to assume that AdventureWorks is moving to a new facility and we want to send out an email to all of our customers and vendors. We want to return a list of names and email addresses to our address update. We can do this in just one query with something like this:

```
USE AdventureWorks2008

SELECT FirstName + ' ' + LastName AS Name, pe.EmailAddress
       EmailAddress
FROM Person.Person pp
JOIN Person.EmailAddress pe
  ON pp.BusinessEntityID = pe.BusinessEntityID
```

```
JOIN Sales.Customer sc
  ON pp.BusinessEntityID = sc.CustomerID

UNION

SELECT FirstName + ' ' + LastName AS Name, pe.EmailAddress
       EmailAddress
FROM Person.Person pp
JOIN Person.EmailAddress pe
  ON pp.BusinessEntityID = pe.BusinessEntityID
JOIN Purchasing.Vendor pv
  ON pp.BusinessEntityID = pv.BusinessEntityID
```

This gets back just one result set:

```
Name                 EmailAddress
-------------------- ----------------------------------------
A. Scott Wright      ascott0@adventure-works.com
Aaron Adams          aaron48@adventure-works.com
Aaron Allen          aaron55@adventure-works.com
…
…
Zachary Wilson       zachary36@adventure-works.com
Zainal Arifin        zainal0@adventure-works.com
Zheng Mu             zheng0@adventure-works.com

(10274 row(s) affected)
```

How It Works

We have our one result set from what would have been two.

SQL Server has run both queries and essentially stacked the results one on top of the other to create one combined result set. Again, notice that the headings for the returned columns all came from the `SELECT` list of the first of the queries.

Moving on to a second example, let's take a look at how a `UNION` deals with duplicate rows — it's actually just the inverse of a normal query in that it assumes you want to throw out duplicates. (In your previous queries, the assumption was that you wanted to keep everything unless you used the `DISTINCT` keyword.) This demo has no real-world potential, but it's quick and easy to run and see how things work.

In this case, we are creating two tables from which we will select. We'll then insert three rows into each table, with one row being identical between the two tables. If our query is performing an `ALL`, then every row (six of them) will show up. If the query is performing a `DISTINCT`, then it will only return five rows (tossing out one duplicate):

```
CREATE TABLE UnionTest1
(
   idcol   int       IDENTITY,
   col2    char(3),
)
```

```
CREATE TABLE UnionTest2
(
   idcol   int       IDENTITY,
   col4    char(3),
)

INSERT INTO UnionTest1
VALUES
   ('AAA')

INSERT INTO UnionTest1
VALUES
   ('BBB')

INSERT INTO UnionTest1
VALUES
   ('CCC')

INSERT INTO UnionTest2
VALUES
   ('CCC')

INSERT INTO UnionTest2
VALUES
   ('DDD')

INSERT INTO UnionTest2
VALUES
   ('EEE')

SELECT col2
FROM UnionTest1

UNION

SELECT col4
FROM UnionTest2

PRINT 'Divider Line--------------------------'

SELECT col2
FROM UnionTest1

UNION ALL

SELECT col4
FROM UnionTest2

DROP TABLE UnionTest1
DROP TABLE UnionTest2
```

Now, look at the heart of what's returned (you'll see some one row(s) affecteds in there — just ignore them until you get to where the results of your query are visible):

```
col2
----
AAA
BBB
CCC
DDD
EEE

(5 row(s) affected)

Divider Line--------------------------
col2
----
AAA
BBB
CCC
CCC
DDD
EEE

(6 row(s) affected)
```

The first result set returned was a simple `UNION` statement with no additional parameters. You can see that one row was eliminated. Even though we inserted "CCC" into both tables, only one makes an appearance since the duplicate record is eliminated by default.

The second return changed things a bit. This time we used a `UNION ALL` and the `ALL` keyword ensured that we get every row back. As such, our eliminated row from the last query suddenly reappears.

Summary

In an RDBMS, the data we want is quite frequently spread across more than one table. `JOIN`s allow us to combine the data from multiple tables in a variety of ways:

- Use an `INNER JOIN` when you want to exclude non-matching fields.
- Use an `OUTER JOIN` when you want to retrieve matches wherever possible, but also want a fully inclusive data set on one side of the `JOIN`.
- Use a `FULL JOIN` when you want to retrieve matches wherever possible, but also want a fully inclusive data set on both sides of the `JOIN`.
- Use a `CROSS JOIN` when you want a Cartesian product based on the records in two tables. This is typically used in scientific environments and when you want to create test data.
- Use a `UNION` when you want the combination of the result of a second query appended to the first query.

There are two different forms of `JOIN` syntax available for `INNER` and `OUTER JOIN`s. I provided the legacy syntax here to help you deal with legacy code, but the newer ANSI format presented through most of this chapter is highly preferable, as it is more readable, is not prone to the ambiguities of the older syntax, and will be supported in SQL Server for the indefinite future.

Over the course of the next few chapters, we will be learning how to build our own tables and *relate* them to each other. As we do this, the concepts of what columns to join on will become even clearer.

Exercises

1. Write a query against the AdventureWorks2008 database that returns one column called `Name` and contains the last name of the employee with `NationalIDNumber` `112457891`.