1

Being Objective: Re-Examining Objects in SQL Server

Well, here we are. We're at the beginning, but, if you're someone who's read my Professional level titles before, you'll find we're not quite at the same beginning that we were at in previous editions. SQL Server has gotten just plain too big, and so the "Pro" is going to become a little bit more "Pro" in level right from the beginning.

So, why am I still covering objects first then? Well, because I think a review is always in order on the basics, because you may need the see what's changed versus previous editions, and, last but not least, I still want this book to retain some of its use as a reference (I hate having to use 25 books to get all the info that seems like it should have been in just one).

With this in mind, I'm going to start off the same way I always start off — looking at the objects available on your server. The difference is that I'm going to assume that you already largely know this stuff, so we're going to move along pretty quickly and make a rather short chapter out of this one.

So, What Exactly Do We Have Here?

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 what kind of data can go into the system, and also facilitating getting data out of the system. If all you want is to tuck the data away somewhere safe, you can 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. In the case of SQL Server, it doesn't just stop there. SQL Server provides additional services that help automate how your data interacts with data from other systems through such powerful features as the SQL Server Agent, Integration Services, Notification Services, and more.

This chapter provides an overview to the rest of the book. Everything discussed in this chapter will be covered again in later chapters, but this chapter is intended to provide you with a roadmap or plan to bear in mind as we progress through the book. As such, in this chapter, we will take a high-level look into:

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

An Overview of Database Objects

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

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 familiar with old versions of SQL Server you may now be saying, "What? What happened to logins? What happened to Remote Servers and 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, you generally don't give them significant thought during the design and programming processes. (They are programmable via something called the SQL Management Objects (SMO), but that is far too special a case to concern you with here. We will look at SMO more fully in Chapter 25.)*

A database is typically a group that includes 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.

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

- master
- model
- msdb
- tempdb

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

- AdventureWorks (the sample database)
- AdventureWorksDW (sample for use with Analysis Services)

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, can, in a pinch, be extremely useful. That said, their direct use is diminishing in importance as Microsoft continues to give more and more other options for getting at system level information.

> **If you're quite cavalier, you may be saying to yourself, "Cool, I can't wait to mess around in there!" *Don't go there!* Using the system tables in any form is fraught with peril. Microsoft has recommended against using the system tables for at least the last three versions of SQL Server. They make absolutely no guarantees about compatibility in the `master` database between versions — indeed, they virtually guarantee that they will change. The worst offense comes when performing updates on objects in the `master` database. Trust me when I tell you that altering these tables in any way is asking for a SQL Server that no longer functions. 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.**

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`.

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. 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 in that not only are the objects within it temporary, but 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 — it 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.*

AdventureWorks

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. (I'll hold off the argument of whether AdventureWorks has the same issue or not. Let's just say that AdventureWorks was, among other things, an attempt to address this problem.) 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.

From the earliest stages of development of Yukon (the internal code name for what we know today as 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. AdventureWorks 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.

I use it as the core sample database for this book.

AdventureWorksDW

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 this.

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 the server writes the changes to the physical database file(s) at a later time.

We'll take a much closer look at how things are logged in Chapter 12, 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 are more central to the make-up of a database than tables. 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 or an update to an existing row, or prevent the deletion of a row.

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 are 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.

Schemas

Schemas provide an intermediate namespace between your database and the other objects it contains. The default namespace in any database is "dbo" (which stands for database owner). Every user has a default schema, and SQL Server will search for objects within that user's default schema automatically. If, however, the object is within a namespace that is not the default for that user, then the object must be referred with two parts in the form of `<schema name>.<object name>`.

> **Schemas replace the concept of "owner" that was used in prior versions of SQL Server. While Microsoft seems to be featuring their use in this release (the idea is that you'll be able to refer to a group of tables by the schema they are in rather than listing them all), I remain dubious at best. In short, I believe they create far more problems than they solve, and I generally recommend against their use (I have made my exceptions, but they are very situational).**

Filegroups

By default, all your tables and everything else about your database (except the log) are stored in a single file. That file is 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 database design, but 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 — or ERD. In an ERD the database is divided into two parts: entities (such as "supplier" and "product") and relations (such as "supplies" and "purchases").

> *Although they have been entirely redesigned with SQL Server 2005, the included database design tools remain a bit sparse. Indeed, the diagramming methodology the tools use doesn't adhere to any of the accepted standards in ER diagramming.*
>
> *Still, these diagramming tools really do provide all the "necessary" things; they are at least something of a start.*

Figure 1-1 is a diagram that shows some of the various tables in the AdventureWorks 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.

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.

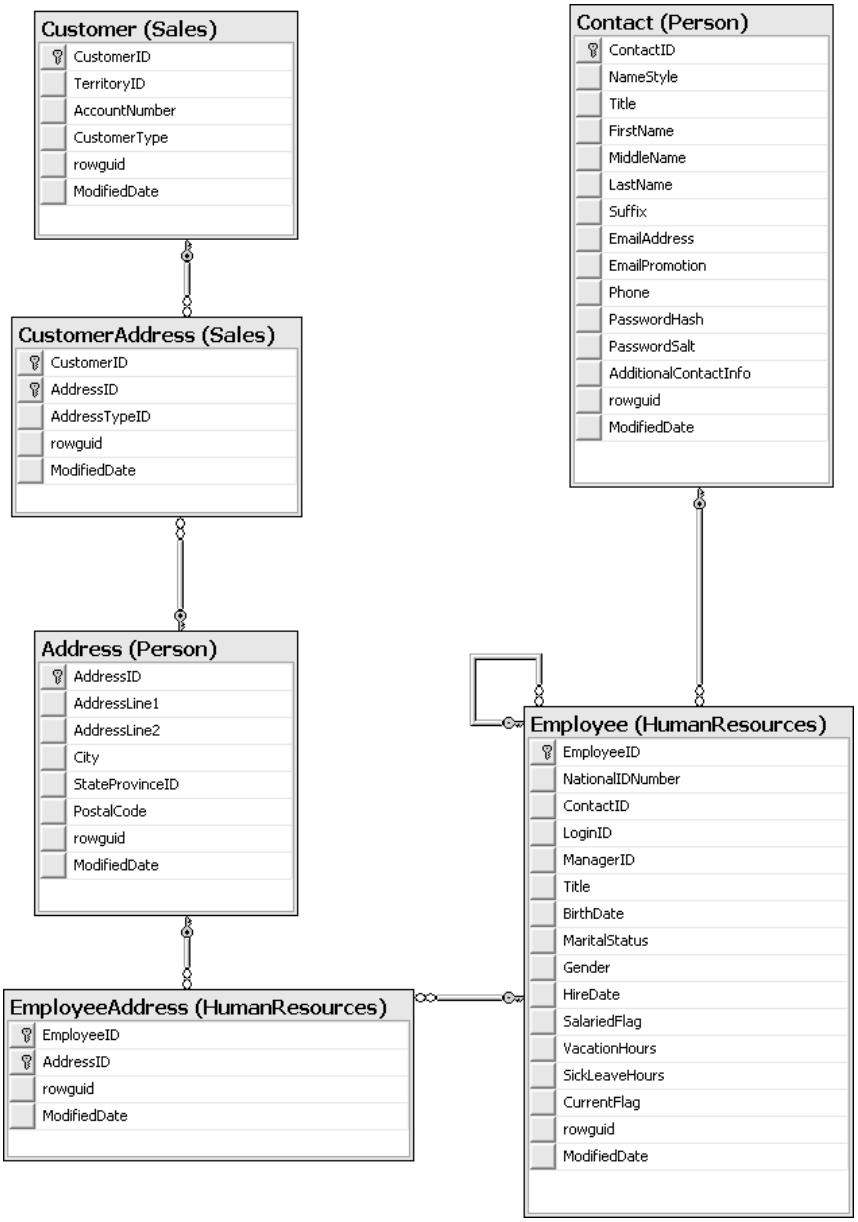

Figure 1-1

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

- Views that reference multiple tables generally perform *much* faster 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 of the index is affected by the update.

We will look into these performance issues more deeply in Chapter 9.

Stored Procedures

Stored procedures (or *sprocs*) are historically and, in the .NET era, even more likely to be 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; as such, 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.

In addition, you can utilize any .NET language to add program constructs, beyond those native to T-SQL, to your stored procedures.

User-Defined Functions

User-defined functions (or *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, however, in that *all* variables passed into the function are passed in by value. If you're familiar with passing in variables `By Ref` in VB, or passing in pointers in C++, 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 11.

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 into 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 rules, constraints aren't really objects unto themselves but rather pieces of metadata describing a particular table.

> **While Microsoft has not stated a particular version for doing so, they have warned that rules will be removed in a future release. Rules should be considered for backward compatibility only and should be avoided in new development. You may also want to consider phasing out any you already have in use in your database.**

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 constraints which are objects, and rules, 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. We will examine both types of defaults in Chapter 5.

> **Much like rules, the form of default that is its own object should be treated as a legacy object and avoided in new development. Use of default constraints is, however, still very valid. See Chapter 5 for more information.**

User-Defined Data Types

User-defined data types are extensions to the system-defined data types. Beginning with this version of SQL Server, 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 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.

> *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. Although these objects are joined at the hip to the tables and columns that they map, they are separate objects and are as such, not automatically updated when changes happen in the database.

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 2005 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 one byte; the next seven 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.
`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.

Table continued on following page

Data Type Name	Class	Size in Bytes	Nature of the Data
`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.
`SmallDateTime`	Date/Time	4	Date and time data from January 1, 1900, to June 6, 2079, with an accuracy of one minute.
`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 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.)
`Unique Identifier`	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.
`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 aren't 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).

Data Type Name	Class	Size in Bytes	Nature of the Data
`Ntext`	Unicode	Varies	Like the Text data type, this is legacy support only. In this case, use `nvarchar(max)`. Variable-length Unicode character data.
`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. Not usable as a data type within a table definition. (You can't nest tables.)
`Sql_variant`	Other	Special	This is loosely related to the `Variant` in VB and C++. Essentially it's a container that enables 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 cast it *explicitly* 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 and the use of special XML-oriented functions.

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 an `int`.

> **SQL Server has no concept of unsigned numeric data types. If you need to allow for larger numbers than the signed data type allows, consider using a larger signed data type. If you need to prevent the use of negative numbers, consider using a `CHECK` constraint that restricts valid data to greater than or equal to zero.**

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 say specifically what type you want to convert to.) A few can't be converted between at all. Figure 1-2 contains a chart that shows the various possible conversions.

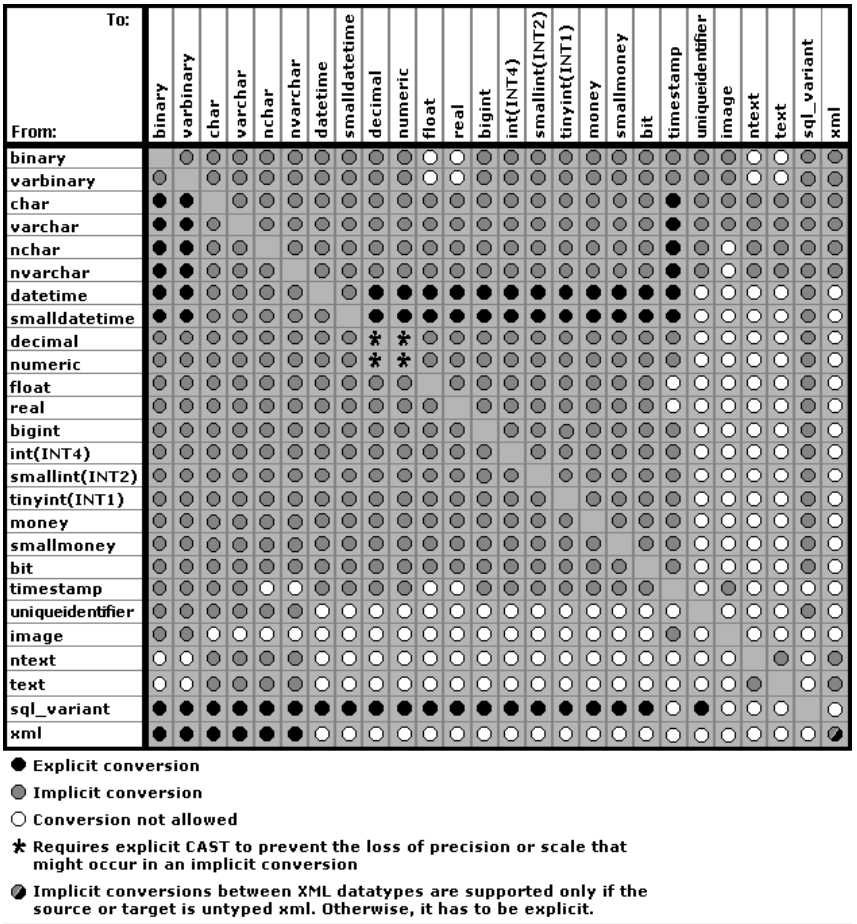

Figure 1-2

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. But 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
Schemas		

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

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 2.0. This includes the letters most westerners are used to — A–Z and a–z. Whether "A" is different from "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 acceptable as a delimiter for column names only if you have `SET QUOTED_IDENTIFIER ON`*. Using square brackets (*`[` *and* `]`*) eliminates the chance that your users will have the wrong setting but is not as platform independent as double quotes are.*

These rules are generally referred to as the rules for identifiers and are in force for any objects you name in SQL Server. Additional rules may exist for specific object types.

Again, I can't stress enough the importance of avoiding the use of SQL Server keywords or embedded spaces in names. Although both are technically legal as long as you qualify them, naming things this way will cause you no end of grief.

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.