1

RDBMS Basics: What Makes Up a SQL Server Database?

COPYRIGHTED MATERIAL

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Understand what the objects are that make up a SQL Server database
- Learn the data types available for use in SQL Server 2012
- Discover how to name objects

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 the 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 I'm 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, you 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, serves as the foundation for this book. 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 you progress through the book. With this in mind, I'll give you 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 (and what not to) call an object 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
- Tables
- Indexes
- Filegroups
- Diagrams
- Views
- Stored procedures
- User-defined functions
- Sequences
- Users
- Roles
- Assemblies
- 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 I'm not going there.) Most, but not all, other objects in a SQL Server are children of the database object.

NOTE *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, you generally won'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.) Although there are some exceptions to this rule, I generally consider them to be advanced in nature, and thus they are not covered in the beginner 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 you store in just one database, and what goes in a separate database? I'll discuss that in some detail later in the book, but for now I'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 have many. Also, any version of SQL Server that you're likely to find in production these days has multiple instances of SQL Server — complete with separate logins and management rights — all on the same physical server. (SQL Server 2000 was already five years old by the time it was replaced, so I'll assume most shops have that or higher.)

NOTE *I'm sure many of you are now asking, "Can I have different versions of SQL Server on the same box — say, SQL Server 2008 and SQL Server 2012?" The answer is yes. You can mix SQL Server 2008 and 2012 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
- **AdventureWorks:** The sample database
- **AdventureWorksDW:** 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`

Because these examples were no longer used in the prior edition of this book, I won't deal with them further here, but I still mention them mostly because they carry fond memories from simpler times, and partly because you might find them out there somewhere.

NOTE *For this edition, the examples will either be homegrown or else come from the newer AdventureWorks samples. The AdventureWorks database is certainly a 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 2012. There is, however, a problem with that — complexity. The AdventureWorks database can sometimes be 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 AdventureWorks 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 developers for years not to use the system tables directly, but, because there were few other options, most developers ignored that advice. Happily, Microsoft began providing other options in the form of system and information schema views; you can now utilize these views to get at the 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.

WARNING *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 want, 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 because 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 points to keep in mind when altering the `model` database:

- **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.
- **Similar pitfalls apply when adding objects or changing settings, which can lead to unintended consequences.** 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 rebuilt from scratch every time you start your SQL Server.

NOTE *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 to which you have access in your system — they will be stored in* `tempdb`*. Creating objects directly in* `tempdb` *gains you nothing, but adding the confusion of referring to things across databases. This is another of those "Don't go there!" kind of things.*

ReportServer

This database will exist only 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, 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 (and only rarely 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.

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. 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 AdventureWorks has the same issues. Let's just say that AdventureWorks was, among other things, an attempt to address this problem.

From the earliest stages of development of SQL Server 2005, Microsoft knew it 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.

AdventureWorks will be something of your home database — you'll use it extensively as you work through the examples in 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 are built. Perhaps the greatest thing about this sample 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 discussed in more detail in Chapters 17 and 18 of this book, and you will be using this database, so keep that 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 available only as a separate download from the Microsoft website. You 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, 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 it available for work with various examples and tutorials you will find on the web.

> **USING NORTHWIND AND PUBS WITH SQL SERVER 2012**
>
> These sample databases are getting a bit long in the tooth, and support has dwindled. You can still use them, but you'll need to go through a conversion process that I'm not going to detail here. If you really want to locate and use these samples, you can do it, but it's going to take a little extra effort.

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).

You'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 and consists of *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.

Take a look at the Production.Location table in the AdventureWorks database. (The view presented in Figure 1-1 is from the SQL Server Management Studio. This is a fundamental tool and you will see 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 14 records. The number of records will go up and down as you 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 you ways of speeding the lookup of your 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 the 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 *user-defined filegroups*. Although there is only one primary filegroup (and it is actually called "Primary"), you can have up to 255 user-defined filegroups. A user-defined filegroup is created as an option to a `CREATE DATABASE` or `ALTER DATABASE` command.

Diagrams

I will discuss database diagramming in some detail when I 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”).

NOTE *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 AdventureWorks database. The diagram also describes many other properties about the database (although it may be a bit subtle since this is new to you). Notice the tiny icons for keys and the infinity sign. These depict the nature of the relationship between two tables. I’ll talk about relationships extensively in Chapters 6 and 8, and I’ll delve further into diagrams later in the book.

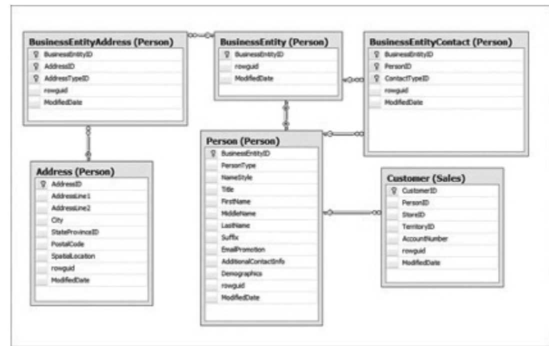

FIGURE 1-2

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 might 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 one or more indexes have been created 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.

You will learn more about 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.

Although 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 (except table variables used as parameters) 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`. I'll examine the impact of this in Chapter 13.

Sequences

Sequences are a new type of object introduced in SQL Server 2012. The job of a sequence is to provide a source of sequential numbers that can be accessed by any number of processes, guaranteeing that no two will retrieve the same next value at the same time. Because they are objects existing on their own — not bound to any table — sequences have a variety of uses that you'll get to look at more closely in Chapter 7.

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, 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 for backward compatibility only, and you should avoid them in new development.

NOTE *Given that Microsoft 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 there are rules, which are objects, and constraints, which are not objects, but metadata). They 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 spatial data types (`Geographic` and `Geometric`) that were added in SQL Server 2008 are implemented using a user-defined type based on a .NET assembly. .NET assemblies are definitely an advanced topic and beyond the scope of this book.

NOTE *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 main database engine and storage mechanisms. Due to their complex nature, full-text indexes are beyond the scope of this text.

SQL SERVER DATA TYPES

Now that you're familiar with the base objects of a SQL Server database, 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 2012 has the intrinsic data types shown in the following Table 1-1:

TABLE 1-1: Data Types

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 quintillion.
`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 (from 1-53, 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.

DATA TYPE NAME	CLASS	SIZE IN BYTES	NATURE OF THE DATA
`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.
`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 nanoseconds (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.

continues

TABLE 1-1 *(continued)*

DATA TYPE NAME	CLASS	SIZE IN BYTES	NATURE OF THE DATA
`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 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 BLOB 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).

DATA TYPE NAME	CLASS	SIZE IN BYTES	NATURE OF THE DATA
`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.
`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. The spatial data types `GEOMETRY` and `GEOGRAPHY` that ship with SQL Server 2012 are implemented as CLR types.

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`.

NOTE *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.

	binary	varbinary	char	varchar	nchar	nvarchar	datetime	smalldatetime	date	time	datetimeoffset	datetime2	decimal	numeric	float	real	bigint	int(INT4)	smallint(INT2)	tinyint(INT1)	money	smallmoney	bit	timestamp	uniqueidentifier	image	ntext	text	sql_variant	xml	CLR UDT	hierarchyid
binary		◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	○	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	○	◐	◐	◐	◐
varbinary	◐		◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	○	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	○	◐	◐	◐	◐
char	●	●		◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	●	◐	◐	◐	◐	◐	◐	◐	◐
varchar	●	●	◐		◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	●	◐	◐	◐	◐	◐	◐	◐	◐
nchar	●	●	◐	◐		◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	●	◐	◐	◐	◐	◐	◐	◐	◐
nvarchar	●	●	◐	◐	◐		◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	●	◐	○	◐	◐	◐	◐	◐	◐
datetime	●	●	◐	◐	◐	◐		◐	◐	◐	◐	◐	●	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	◐	○	○	○
smalldatetime	●	●	◐	◐	◐	◐	◐		◐	◐	◐	◐	●	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	◐	○	○	○
date	●	●	◐	◐	◐	◐	◐	◐		○	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○
time	●	●	◐	◐	◐	◐	◐	◐	○		◐	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○
datetimeoffset	●	●	◐	◐	◐	◐	◐	◐	◐	◐		◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○
datetime2	●	●	◐	◐	◐	◐	◐	◐	◐	◐	◐		○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○
decimal	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	✱	✱	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	○	○	○
numeric	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	✱	✱	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	○	○	○
float	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐		◐	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	○	○	○
real	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐	◐		◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	○	○	○
bigint	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐		◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	○	○	○
int(INT4)	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	◐		◐	◐	◐	◐	◐	◐	○	○	○	○	◐	○	○	○
smallint(INT2)	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	◐	◐		◐	◐	◐	◐	◐	○	○	○	○	◐	○	○	○
tinyint(INT1)	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	◐	◐	◐		◐	◐	◐	◐	○	○	○	○	◐	○	○	○
money	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	◐	◐	◐	◐		◐	◐	◐	○	○	○	○	◐	○	○	○
smallmoney	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	◐	◐	◐	◐	◐		◐	◐	○	○	○	○	◐	○	○	○
bit	◐	◐	◐	◐	◐	◐	◐	◐	○	○	○	○	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐		◐	○	○	○	○	◐	○	○	○
timestamp	◐	◐	◐	◐	○	○	◐	◐	○	○	○	○	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐		○	◐	○	○	○	○	○	○
uniqueidentifier	◐	◐	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○	○	○	◐	○	○	○
image	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○		○	○	○	○	○	○
ntext	○	○	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○	○	◐	○	○
text	○	○	◐	◐	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐		○	◐	○	○
sql_variant	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	○	○	○		○	○	○
xml	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		◐	○
CLR UDT	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○
hierarchyid	●	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○

● Explicit conversion
◐ Implicit conversion
○ Conversion not allowed
✱ Requires explicit CAST to prevent the loss of precision or scale that might occur in an implicit conversion.
⊘ Implicit conversions between xml data types are supported only if the source or target is untyped xml. Otherwise, the conversion must be explicit.

FIGURE 1-3

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

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

Code snippet Chap01.SQL

NOTE *I 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.
```

Not exactly what you were after, is it? Now 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 01/01/2012

(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 you'll run into these conversions on a regular basis. Fortunately, the `CAST` and `CONVERT()` functions enable you to convert between many SQL Server data types. I 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 you 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 you know. In short, it means *I don't know*, *undefined,* or perhaps *not applicable*.

SQL SERVER IDENTIFIERS FOR OBJECTS

Now you've heard all sorts of things about objects in SQL Server. It's time to 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
- Sequences
- 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 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 *Note that double quotes are acceptable as a delimiter for column names only 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.

NOTE *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 either of these things! Columns with embedded spaces in their names 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 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.

EXERCISES

1. What is the purpose of the `master` database?
2. What are two differences between the `datetime` and `datetime2` data types?

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	CONCEPT
Types of Objects	A new installation of SQL Server creates system databases and permits creation of many types of objects both within and external to those databases.
Data Types	SQL Server provides a variety of data types that can be used to efficiently and correctly store information.
Identifiers	You can give your objects names up to 128 characters (116 for temporary objects) that start with a letter.