CHAPTER 3

PHYSICAL ELEMENTS OF DATA MODELS

Now that you have a grasp of the logical elements used to construct a data model, let's look at the physical elements. These are the objects that you use to build the database. Most of the objects you build into your physical model are based on objects you created in the logical model. Many physical elements are the same no matter which RDBMS you are using, but we look at all the elements available in SQL Server 2008. It is important to know SQL Server's capabilities so that you can build your model with them in mind.

In this chapter, we cover all the physical SQL Server objects in detail and walk you through how to use each type of object in your physical model. You will use these elements later in Chapter 9.

Physical Storage

First, we'll start with the objects that allow you to store data in your database. You'll build everything else on these objects. Specifically, these are tables, views, and data types.

Tables

Tables are the building blocks on which relational databases are built. Underneath everything else, all data in your database ends up in a table. Tables are made up of rows and columns. Like a single instance in an entity, each row stores information pertaining to a single record. For example, in an employee table, each row would store the information for a single employee.

The columns in the table store information about the rows in the table. The FirstName column in the Employee table would store the first names of all the employees. Columns map to attributes from your logical model, and, like the logical model, each column has a data type assigned. Later in this chapter we look at the SQL Server data types in detail.

When you add data to a table, each column must either contain data (even if it is an empty string) or specify a NULL value, NULL being the complete absence of data. Additionally, you can specify that each column have a default value. The **default value** is used if you add data without specifying a value for that column. A default can be a fixed value, such as always setting a numeric column to the value of 12, or it can be a function that returns a value of the appropriate data type. If you do not have a default value specified and you insert data without specifying a value for a column, SQL Server attempts to insert a NULL value. If the column does not allow NULL values, your insert will fail.

You can think of a table as a single spreadsheet in an application such as Microsoft Excel. In fact, an Excel spreadsheet is a table, but Excel is not a relational database management system. A database is really nothing more than a collection of tables that store information. Sure, there are many other objects in a database, but without tables you would not have any data. Using Transact-SQL, also known as T-SQL, you can manipulate the data in a table. The four basic Data Manipulation Language (DML) statements are defined as follows:

- SELECT: Allows users to retrieve data in a table or tables
- INSERT: Allows users to add data to a table
- UPDATE: Allows users to change data in a table
- DELETE: Allows users to remove data from a table

How SQL Server Stores Tables

In addition to understanding what tables are, it's important that you understand how SQL Server stores them; the type of data your columns store will dictate how the table is stored on disk, and this can directly affect the performance of your database. Everything in SQL Server is stored on *pages*. **Pages** are 8K contiguous allocations of information on the disk, and there are different kinds of pages depending on what is on the page. For our purposes, we will focus on **data pages**: pages that store table data. Each row you add to a table is stored on a page, and depending on the size of the data in the row, the row can be stored either on a page with other rows, or on its own page or pages.

Before SQL Server 2005, data and overhead for a single row could not exceed 8,060 bytes (8K). This was a hard limit that you had to account for when designing tables. In SQL Server 2005, this limit has been overcome, in a manner of speaking. Now, if your row exceeds 8,060 bytes, SQL Server moves one or more of your variable-length columns onto a new page and leaves a 24-byte pointer in its place. This does not mean that you have an unlimited row size, nor should you make all your rows bigger than 8,060 bytes. Why not? First, notice that we said SQL Server will move *variable-length* columns. This means that you are still limited to 8,060 bytes of *fixed-length* columns. Additionally, you are still limited to 8K on your primary data page for the row. Remember the 24-byte pointer we mentioned? In theory you are limited to around 335 pointers on the main page. As ridiculous as a 336-column varchar(8000) table may sound, we have seen far stranger.

If SQL Server manages all this behind the scenes, why should you care? Here's why. Although SQL Server moves the variable-length fields to new pages after you exceed the 8K limit, the result is akin to a fragmented hard drive. You now have chunks of data that need to be assembled when accessed, and this adds processing time. As a data modeler you should always try to keep your rows smaller than the 8K limit for performance reasons. There are a few exceptions to this rule, and we look at them more closely later in this chapter when we discuss data types. Keep in mind that there is a lot more complexity in the way SQL Server handles storage and pages than we cover here, but your data model can't affect the other variables as much as it can affect table size.

Views

Views are simply stored T-SQL that uses SELECT statements to display data from one or more tables. The tables referenced by views are often referred to as the view's **base tables.** Views, as the name implies, allow you to create various pictures of the underlying information. You can reference as many or as few columns from each base table as you need to make your views. This capability allows you to slice up data and display only relevant information.

You access views in almost the same way that you access tables. All the basic DML statements work against views in the same way they do on tables, with a few exceptions. If you have a view that references more than one base table, you can use only INSERT, UPDATE, or DELETE statements that reference columns from one base table. For example, let's assume that we have a view that returns customer data from two tables. One table stores the customer's information, and the other holds the address data for that customer. The definition of the customer_address view is as follows:

```
CREATE VIEW customer_address
AS
SELECT customer.first_name,
     customer.last_name,
     customer.phone,
     address.address_line1,
     address.city,
     address.state,
     address.zip
FROM customer
JOIN address
     ON address.customer_id = customer.customer_id
WHERE address.type = 'home'
```

You can perform INSERT, UPDATE, and DELETE operations against the customer_address view as long as you reference only the customer table *or* the address table.

You may be asking yourself, "Why would I use a view instead of just referencing the tables directly?" There are several reasons to use views in your database. First, you can use a view to obscure the complexity of the underlying tables. If you have a single view that displays customer and address information, developers or end users can access the information they need from the view instead of needing to go to both tables. This technique eliminates the need for users to understand the entire database; they can focus on a single object. You gain an exponential benefit when you start working with many base tables in a single view.

Using views also allows you to change the tables or the location where the data is stored without affecting users. In the end, as long as you update the view definition so that it accommodates the table changes you made, your users will never need to know that there was a change. You can also use views to better manage security. If you have users who need to see some employee data but not sensitive data such as social security numbers or salary, you can build a view that displays only the information they need.

Finally, consider how using views can save you time when querying your database. Every time you run T-SQL code, SQL Server must first compile the code. This transforms the human-readable SELECT statement into a form that the SQL Server engine can understand, and the resulting code is an **execution plan.** Execution plans for running views are stored in SQL Server, and the T-SQL code behind them is compiled. This process takes time, but with views, the compilation is done only when the view is created. This saves you processing each time you call the view. The first time a view is called, SQL Server figures out the best way to retrieve the data from the base tables, given the table structure and the indexes in place. This execution plan is cached and reused the next time the view is called.

In our humble opinion, views are probably the most underused feature in SQL Server. For some reason, people tend to avoid the use of views or use them in inefficient ways. In Chapter 11 we look at some of the most beneficial uses for views.

Data Types

As mentioned earlier, every column in each of your tables must be configured to store a specific type of data. You do this by associating a data type with the column. Data types are what you use to specify the type, length, precision, and scale of data that can be stored in the column. SQL Server 2008 gives you several general categories of data types, with each category containing specific data types. Many of these data types are similar to the types we looked at in Chapter 2. In this section, we look at each of the SQL Server data types and talk about how the SQL Server engine handles and stores them.

When you build your model, it is important to understand how much space each data type requires. The difference between a data type that needs 2 bytes versus one that requires 4 bytes may seem insignificant, but when you multiply the extra 2 bytes over millions or billions of rows, you could end up needing tens or hundreds of gigabytes of additional storage.

SQL Server 2008 has functionality (parts of which were introduced in SQL Server 2005 Service Pack 2) that allows the SQL Server storage engine to compress data at the row and page levels. However, this functionality is limited to the Enterprise Edition and is, in general, more of an administrative concern. Your estimate of data storage requirements, which is based on the numbers we talk about here, should be limited to the uncompressed storage requirements. Enabling data compression in a database is something that a database administrator will work on with the database developer after the database has been built. With that said, let's look at the data types available in SQL Server 2008.

Numeric Data Types

Our databases need to store many kinds of numbers that we use day to day. Each of these numbers is unique and requires us to store varying pieces of data. These differences in numbers and requirements dictate that SQL Server be able to support 11 numeric data types. Following is a review of all the numeric data types available in SQL Server. Also, Table 3.1 shows the specifications on each numeric data type.

Table 3.1 Numeric Data Type Specifications

Data Type	Value Range	Storage
bigint	–9,223,372,036,854,775,808 through 9,223,372,036,854,775,807	8 bytes
bit	0 or 1	1 byte (minimum)
decimal	Depends on precision and scale	5–17 bytes
float	–1.79E+308 through –2.23E–308, 0, and 2.23E–308 through 1.79E+308	4 or 8 bytes
int	–2,147,483,648 to 2,147,483,647	4 bytes
money	–922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
numeric	Depends on precision and scale	5–17 bytes
real	–3.40E+38 to –1.18E–38, 0, and 1.18E–38 to 3.40E+38	4 bytes
smallint	–32,768 to 32,767	2 bytes
smallmoney	–214,748.3648 to 214,748.3647	4 bytes
tinyint	0 to 255	1 byte

Int

The int data type is used to store whole integer numbers. Int does not store any detail to the right of the decimal point, and any number with decimal data is rounded off to a whole number. Numbers stored in this type must be in the range of –2,147,483,648 through 2,147,483,647, and each piece of int data requires 4 bytes to store on disk.

Bigint

Bigint is just what it sounds like: a big integer number. When you need larger numbers than supported by the int data type, you can use bigint. Using bigint expands your range from the paltry 2 billion of an int and allows you to store numbers from approximately negative 9 quintillion all the way to 9 quintillion. (A quintillion is a 1 followed by 18 zeros.) Bigger numbers require more storage; bigint data requires 8 bytes.

Smallint

On the other side of the int data type, we have smallint. Smallint can hold numbers from –32,768 through 32,767 and requires only 2 bytes of storage.

Tinyint

Rounding out the int family of data types is the tinyint. Requiring only 1 byte of storage and capable of storing numbers from 0 through 255, tinyint is perfect for status columns. Note that tinyint is the only int data type that cannot store negative numbers.

Bit

The bit data type is the SQL Server equivalent of a flag or a Boolean. The only valid values are 0, 1, or NULL, making the bit data type perfect for storing on or off, yes or no, or true or false. Bit storage is a bit more complex (pardon the pun). Storing a 1 or a 0 requires only 1 bit on disk, but the minimum storage for bit data is 1 byte. For any given table, the bit columns are lumped together for storage. This means that when you have 1-bit to 8-bit columns they collectively take up 1 byte. When you have 9- to 16-bit columns, they take up 2 bytes, and so on. SQL Server implicitly converts the strings TRUE and FALSE to bit data of 1 and 0, respectively.

Decimal and Numeric

In SQL Server 2008, the decimal and numeric data types are exactly the same. Previous versions of SQL Server do not have a numeric data type; it was added in SQL Server 2005 so that the terminology would fall in line with other RDBMS software. Both these data types hold numbers complete with detail to the right of the decimal. When using decimal or numeric, you can specify a precision and a scale. Precision sets the total number of digits that can be stored in the number. Precision can be set to any value from 1 through 38, allowing decimal numbers to contain 1 through 38 digits. Scale specifies how many of the total digits can be stored to the right of the decimal point. Scale can be any number from 0 to the precision you have set. For example, the number 234.67 has a precision of 5 and a scale of 2. The storage requirements for decimal and numeric vary depending on the precision. Table 3.2 shows the storage requirements based on precision.

Table 3.2 Decimal and Numeric Storage Requirements

Precision	Storage
1 through 9	5 bytes
10 through 19	9 bytes
20 through 28	13 bytes
29 through 38	17 bytes

Money and Smallmoney

Both the money and the smallmoney data types store monetary values to four decimal places. The only difference in these two types is that money can store values from about –922 trillion through 922 trillion and requires 8 bytes of storage, whereas smallmoney holds only values of –214,748.3648 through 214,748.3647 and requires only 4 bytes of storage. Functionally, these types are similar to decimal and numeric, but money and smallmoney also store a currency symbol such as $ (dollar), ¥ (yen), or £ (pound).

Float and Real

Both float and real fall into the category of approximate numbers. Each holds values in scientific notation, which inherently causes data loss because of a lack of precision. If you don't remember your high school chemistry class, we briefly explain scientific notation. You basically store a small subset of the value, followed by a designation of how many decimal places should precede or follow the value. So instead of storing 1,234,467,890 you can store it as 1.23E+9. This says that the decimal in 1.23 should be moved 9 places to the right to determine the actual number. As you can see, you lose a lot of detail when you store the number in this way. The original number (1,234,467,890) becomes 1,230,000,000 when converted to scientific notation and back.

Now back to the data types. Float and real store numbers in scientific notation; the only difference is the range of values and storage requirements for each. See Table 3.1 for the range of values for these types. Real requires 4 bytes of storage and has a fixed precision of 7. With float data, you can specify the precision or the total number of digits, from 1 through 53. The storage requirement varies from 4 bytes (when the precision is less than 25) to 8 bytes (when the precision is 25 through 53).

Date and Time Data Types

When you need to store a date or time value, SQL Server provides you with six data types. Knowing which type to use is important, because each date and time data type provides a slightly different level of accuracy, and that can make a huge difference when you're calculating exact times, as well as durations. Let's look at each in turn.

Datetime and Smalldatetime

The datetime and smalldatetime data types can store date and time data in a variety of formats; the difference is the range of values that each can store. Datetime can hold values from January 1, 1753, through December 31, 9999, and can be accurate to 3.33 milliseconds. In contrast, smalldatetime can store dates only from January 01, 1900, through June 6, 2079, and is accurate only to 1 minute. For storage, datetime requires 8 bytes, and smalldatetime needs only 4 bytes.

Date and Time

New in SQL Server 2008 are data types that split out the date portion and the time portion of a traditional date and time data type. Literally, as the names imply, these two data types account for either the date portion (month, day, and year), or the time portion (hours, minutes, seconds, and nanoseconds). Thus, if needed, you can store only one portion or the other in a column.

The range of valid values for the date data type are the same as for the datetime data type, meaning that date can hold values from January 1, 1753, through December 31, 9999. From a storage standpoint, date requires only 3 bytes of space, with a character length of 10.

The time data type holds values 00:00:00.0000000 through 23:59:59.9999999 and can hold from 8 characters (hh:mm:ss) to 16 characters (hh:mm:ss:*nnnnnnn*), where *n* represents fractional seconds. For example, 13:45:25.5 literally means that it is 1:45:25 and one-half second p.m. You can specify the scale of the time data type from 0 to 7 to designate how many digits you can use for fractional seconds. At its maximum, the time data type requires 5 bytes of storage.

Datetime2

Another new data type in SQL Server 2008 is the datetime2 data type. This is very similar to the original datetime data type, except that datetime2 incorporates the precision and scale options of the time data type. You can specify the scale from 0 to 7, depending on how you want to divide and store the seconds. Storage for this data type is fixed at 8 bytes, assuming a precision of 7.

Datetimeoffset

The final SQL Server 2008 date and time data type addition is datetimeoffset. This is a standard date and time data type, similar to datetime2 (because it can store the precision). Additionally, datetimeoffset can store a plus or minus 14-hour offset. It is useful in applications where you want to store a date and a time along with a relative offset, such as when you're working with multiple time zones. The storage requirement for datetimeoffset is 10 bytes.

String Data Types

When it comes to storing string or character data, the choice and variations are complex. Whether you need to store a single letter or the entire text of *War and Peace,* SQL Server has a string data type for you. Fortunately, once you understand the difference between the available string data types, choosing the correct one is straightforward.

Char and Varchar

Char and varchar are probably the most used of the string data types. Each stores standard, non-Unicode text data. The differences between the two lie mostly in the storage of the data. In each case, you must specify a length when defining a column as char or varchar. The length sets the limit on the number of characters the column can hold.

Here's the kicker: The char data type always requires the same number of bytes for storage as you have specified for the length. If you have a char(20), it will always require 20 bytes of storage, even if you store only a 5-character word in the column. With a varchar, the storage is always the actual number of characters you have stored plus 2 bytes. So a varchar(20) with a 5-character word will take up 7 bytes, with the extra 2 bytes holding a size reference for SQL Server. Each type can have a length of as many as 8,000 characters.

When do you use one over the other? The rule of thumb is to use char when all the data will be close to the same length, and use varchar when the data will vary a great deal. Following this rule should make for optimum storage.

Another tip is to avoid using varchar for short columns. We have seen databases use varchar(2) columns, and the result is wasted space. Let's assume you have 100 rows in your table and the table contains a varchar(2) column. Assuming all the columns are NULL, you still need to store the 2 bytes of overhead, so without storing any data you have already taken up as much space as you would using char(2).

One other special function of varchar is the **max** length option. When you specify max as the length, your varchar column can store as much as 2^31–1 bytes of data, which is about 2 trillion bytes, or approximately 2GB of string data. If you don't think that's a lot, open your favorite text editor and start typing until you reach a 2GB file. Go on, we'll wait. It's a lot of information to cram into a single column. Varchar(max) was added to SQL Server in the 2005 release and was meant to replace the text data type from previous versions of SQL Server.

Nchar and Nvarchar

The nchar and nvarchar data types work in much the same way as the char and varchar data types, except that the *n* versions store Unicode data. Unicode is most often used when you need to store non-English language strings that require special characters such as the Greek letter beta (β). Because Unicode data is a bit more complex, it requires 2 bytes for each character, and thus an nchar requires double the length in bytes for storage, and nvarchar requires double the actual number of characters plus the obligatory 2 bytes of overhead.

From our earlier discussion, recall that SQL Server stores tables in 8,060-byte pages. Well, a single column cannot span a page, so some simple math tells us that when using these Unicode data types, you will reach 8,000 bytes when you have a length of 4,000. In fact, that is the limit for the nchar and nvarchar data types. Again, you can specify nvarchar(max), which in SQL Server 2005 replaced the old ntext data type.

Binary and Varbinary

Binary and varbinary function in exactly the same way as char and varchar. The only difference is that these data types hold binary information such as files or images. As before, varbinary(max) replaces the old image data type. In addition, SQL Server 2008 allows you to specify the filestream attribute of a varbinary(max) column, which switches the storage of the BLOB. Instead of being stored as a separate file on the file system, it is stored in SQL Server pages on disk.

Text, Ntext, and Image

As mentioned earlier, the text, ntext, and image data types have been replaced with the max length functionality of varchar, nvarchar, and varbinary, respectively. However, if you are running on an older version or upgrading to SQL Server 2005 or SQL Server 2008, you may still need these data types. The text data type holds about 2GB of string data, and ntext holds about 1GB of Unicode string data. Image is a variable-length binary field and can hold any binary data, up to about 2GB. When using these data types, you must use certain functions to write, update, and read to the columns; you cannot just do a simple update. Keep in mind that these three data types have been replaced, and Microsoft will likely remove them from future releases of SQL Server.

Other Data Types

In addition to the standard numeric and string data types, SQL Server 2008 provides several other useful data types. These additional types allow you to store XML data, **g**lobally **u**nique **id**entifiers (GUIDs), hierarchical identities, and spatial data types. There is also a new file storage data type that we'll talk about shortly.

Sql_variant

A column defined as sql_variant can store most any data that can be stored in the other SQL Server data types. The only data you cannot put into a sql_variant are text, ntext, image, xml, timestamp, or the max length data types. Using sql_variant you can store various data types in the same column of a table. As you will read in Chapter 4, this is not the best practice from a modeling standpoint. That said, there are some good uses for sql_variant, such as building a staging table when you're loading less-than-perfect data from other sources. The storage requirement for a sql_variant depends on the type of data you put in the column.

Timestamp

This data type has a somewhat misleading name. In fact timestamp does not store any sort of time or date information. Instead, timestamp is a binary number that is automatically incremented each time an insert or update happens to a table containing the timestamp column. The counter for the timestamp column is stored for the entire database, and each table is allowed to have only a single timestamp column. In this way, you can tell in what order various operations have happened in your database, or you can implement row versioning.

We once used timestamp to archive a large database. Each night we would run a job to grab all the rows from all the tables where the timestamp was greater than the last row copied the night before. Timestamps require 8 bytes of storage, and remember, 8 bytes can add up fast if you add timestamps to all your tables.

Uniqueidentifier

The uniqueidentifier data type is probably one of the most interesting data types available, and it is the topic of much debate. Basically, a uniqueidentifier column holds a GUID—a string of 32 random characters in blocks separated by hyphens. For example, the following is a valid GUID:

```
45E8F437-670D-4409-93CB-F9424A40D6EE
```

Why would you use a uniqueidentifier column? First, when you generate a GUID, it will be a completely unique value and no other GUID in the world will share the same string. This means that you can use GUIDs as PKs on your tables if you will be moving data between databases. This technique prevents duplicate PKs when you actually copy data.

When you're using uniqueidentifier columns, keep in mind a couple of things. First, they are pretty big, requiring 16 bytes of storage. Second, unlike timestamps or identity columns (see the section on primary keys later in this chapter), a uniqueidentifier does not automatically have a new GUID assigned when data is inserted. You must use the NEWID function to generate a new GUID when you insert data. You can also make the default value for the column NEWID(). In this way, you need not specify anything for the uniqueidentifier column; SQL Server will insert the GUID for you.

Xml

The xml data type is a bit outside the scope of this book, but we'll say a few words about it. Using the xml data type, SQL Server can hold Extensible Markup Language (XML) data in a column. Additionally, you can bind an XML schema to the column to constrain the XML data being stored. Like the max data types, the xml data type is limited to 2GB of storage.

Table

A table data type can store the result set of T-SQL statements for processing later. The data is stored in a similar fashion to the way an entire table is stored. It is important to note that the table data type *cannot* be used on columns; it can be used only in variables in T-SQL code. Programming in SQL Server is beyond the scope of this book, but the table data type plays an important role in user-defined functions, which we discuss shortly.

Table variables behave in the same way as base tables. They contain columns and can have check constraints, unique constraints, and primary keys. As with base tables, a table variable can be used in SELECT, INSERT, UPDATE, and DELETE statements. Like other local variables, table variables exist in the scope of the calling function and are cleaned up when the calling module finishes executing. To use table variables, you declare them like any other variable and provide a standard table definition to the declaration.

Hierarchyid

The hierarchyid data type is a system-provided data type that allows you to store hierarchical data, such as organizational data, project tasks, or file system–style data in a relational database table. Whenever you have self-referencing data in a tiered format, hierarchyid allows you to store and query the data more efficiently. The actual data in a hierarchyid is represented as a series of slashes and numerical designations. This is a specialized data type and is used only in very specific instances.

Spatial Data Types

SQL Server 2008 also introduces the spatial data types for relational storage. The first of the two new data types is geometry, which allows you to store planar data about physical locations (distances, vectors, etc.). The other data type, geography, allows you to store round earth data such as latitude and longitude coordinates. Although this is oversimplifying, these data types allow you to store information that can help you determine the distance between locations and ways to navigate between them.

User-Defined Data Types

In addition to the data types we have described, SQL Server allows you to create user-defined data types. With **user-defined data types,** you can create standard columns for use in your tables. When defining user-defined data types, you still must use the standard data types that we have described here as a base. A user-defined data type is really a fixed definition of a data type, complete with length, precision, or scale as applicable.

For example, if you need to store phone numbers in various tables in your database, you can create a phone number data type. If you create the phone number data type as a varchar(25), then every column that you define as a phone number will be exactly the same, a varchar(25). As you recall from the discussion of domains in Chapter 2, user-defined data types are the physical implementation of domains in SQL Server. We highly recommend using user-defined data types for consistency, both during the initial development and later during possible additions to your data model.

Referential Integrity

We discussed referential integrity (RI) in Chapter 2. Now we look specifically at how you implement referential integrity in a physical database.

In general, data integrity is the concept of keeping your data consistent and helping to ensure that your data is an accurate representation of the real world and that it is easy to retrieve. There are various kinds of integrity; referential integrity ensures that the relationships between tables are adhered to when you insert or update data. For example, suppose you have two tables: one called Employee and one called Vehicle. You require that each vehicle be assigned to an employee; this is done via a relationship, and the rule is maintained with RI. You physically implement this relationship using primary and foreign keys.

Primary Keys

A primary key constraint in SQL Server works in the same way as a primary key does in your logical model. A primary key is made up of the column or columns that uniquely identify the row in any given table.

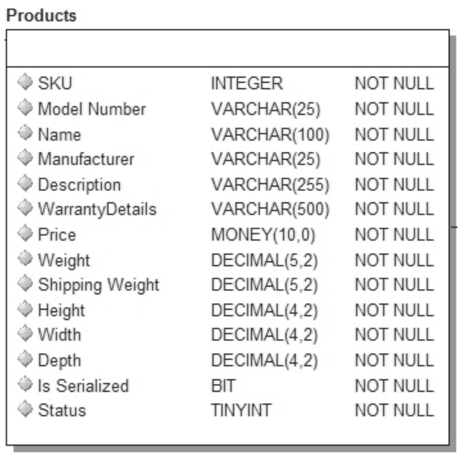
The first step in creating a PK is to identify the columns on which to create the key; most of the time this is decided during logical modeling. What makes a good primary key in SQL Server, and, more importantly, what makes a poor key? Any column or combination of columns in your table that can uniquely identify the row are known as **candidate keys.** Often there are multiple candidate keys in a table. Our first tip for PK selection is to avoid string columns. When you join two tables, SQL Server must compare the data in the primary key to the data in the other table's foreign key. By their nature, strings take more time and processing power to compare than do numeric data types.

That leaves us with numeric data. But what kind of numeric should you use? Integers are always good candidates, so you could use any of the int data types as long as they are large enough to be unique given the table's potential row count. Also, you can create a composite PK (a PK that uses more than one column), but we do not recommend using composite PKs if you can avoid it. The reason? If you have four columns in your PK, then each table that references this table will require the same four columns. Not only does it take longer to build a join on four columns, but also you have a lot of duplicate data storage that would otherwise be avoided.

To recap, here are the rules you should follow when choosing a PK from your candidate keys.

- Avoid using string columns.
- Use integer data when possible.
- Avoid composite primary keys.

Given these rules, let's look at a table and decide which columns to use as our PK. Figure 3.1 shows a table called Products. This table has a couple of candidate keys, the first being the model number. However, model numbers are unique only to a specific manufacturer. So the best option here would be a composite key containing both Model Number and Manufacturer. The other candidate key in this table is the SKU. An SKU (stock-keeping unit) number is usually an internal number that can uniquely identify any product a company buys and sells regardless of manufacturer.



Products

SKU	INTEGER	NOT NULL
Model Number	VARCHAR(25)	NOT NULL
Name	VARCHAR(100)	NOT NULL
Manufacturer	VARCHAR(25)	NOT NULL
Description	VARCHAR(255)	NOT NULL
WarrantyDetails	VARCHAR(500)	NOT NULL
Price	MONEY(10,0)	NOT NULL
Weight	DECIMAL(5,2)	NOT NULL
Shipping Weight	DECIMAL(5,2)	NOT NULL
Height	DECIMAL(4,2)	NOT NULL
Width	DECIMAL(4,2)	NOT NULL
Depth	DECIMAL(4,2)	NOT NULL
Is Serialized	BIT	NOT NULL
Status	TINYINT	NOT NULL

Figure 3.1 A Products table in need of a primary key

Let's look at each of the candidates and see whether it violates a rule. The first candidate (Model Number and Manufacturer) violates all the rules; the data is a string, and it would be a composite key. So that leaves us with SKU, which is perfect; it identifies the row, it's an integer, and it is a single column.

Now that we have identified our PK, how do we go about configuring it in SQL Server? There are several ways to make PKs, and the method you use depends on the state of the table. First, let's see how to do it at the same time you create the table. Here is the script to create the table, complete with the PK.

```
CREATE TABLE Products(
    sku               int             NOT NULL   PRIMARY KEY,
    modelnumber       varchar(25)     NOT NULL,
    name              varchar(100)    NOT NULL,
    manufacturer      varchar(25)     NOT NULL,
    description       varchar(255)    NOT NULL,
    warrantydetails   varchar(500)    NOT NULL,
    price             money           NOT NULL,
    weight            decimal(5, 2)   NOT NULL,
    shippingweight    decimal(5, 2)   NOT NULL,
    height            decimal(4, 2)   NOT NULL,
    width             decimal(4, 2)   NOT NULL,
    depth             decimal(4, 2)   NOT NULL,
    isserialized      bit             NOT NULL,
    status            tinyint         NOT NULL
)
```

You will notice the `PRIMARY KEY` statement following the definition of the sku column. That statement adds a PK to the table on the sku column, something that is simple and quick.

However, this method has one inherent problem. When SQL Server creates a PK in the database, every PK has a name associated with it. Using this method, we don't specify a name, so SQL Server makes one up. In this case it was PK_Products_30242045. The name is based on the table name and some random numbers. On the surface, this doesn't seem to be a big problem, but what if you later need to delete the PK from this table? If you have proper change control in your environment, then you will create a script to drop the key and you will drop the key from a quality assurance server first. Once tests confirm that nothing else will break when this key is dropped, you go ahead and run the script in production. The problem is that if you create the table using the script shown here, the PK will have a different name on each server and your script will fail.

How do you name the key when you create it? What you name your keys is mostly up to you, but we provide some naming guidelines in Chapter 7. In this case we use pk_product_sku as the name of our PK. As a best practice, we suggest that you always explicitly name all your primary keys in this manner. In the following script we removed the `PRIMARY KEY` statement from the sku column definition and added a `CONSTRAINT` statement at the end of the table definition.

```
CREATE TABLE Products(
    sku                 int                 NOT NULL,
    modelnumber         varchar(25)         NOT NULL,
    name                varchar(100)        NOT NULL,
    manufacturer        varchar(25)         NOT NULL,
    description         varchar(255)        NOT NULL,
    price               money               NOT NULL,
    weight              decimal(5, 2)       NOT NULL,
    shippingweight      decimal(5, 2)       NOT NULL,
    height              decimal(4, 2)       NOT NULL,
    width               decimal(4, 2)       NOT NULL,
    depth               decimal(4, 2)       NOT NULL,
    isserialized        bit                 NOT NULL,
    status              tinyint             NOT NULL,
CONSTRAINT pk_product_sku PRIMARY KEY (sku)
)
```

Last, but certainly not least, what if the table already exists and you want to add a primary key? First, you must make sure that any data already in the column conforms to the rules of a primary key. It cannot contain NULLs, and each row must be unique. After that, another simple script will do the trick.

```
ALTER TABLE Products
ADD CONSTRAINT pk_product_sku PRIMARY KEY (sku)
```

But wait—there's more. Using the sku column as we have done here is fine, but there are other PK options we need to discuss. If you were to go through your entire database and define PKs as we have done on the Products table, you would likely end up with a different column name in each table that holds the primary key. This is not necessarily a bad thing, but it means that you must look up the data type and column name whenever you want to add another column with a foreign key or you need to write a piece of code to join tables.

Wouldn't it be nice if all your tables had their PKs in columns having the same name? For example, every table in your database could be given a column named objectid and that column could simply have an arbitrary unique integer. In this case, you can use an identity column in SQL Server to manage your integer PK value. An **identity column** is one that automatically increments a number with each insert into the table. When you make a column an identity, you specify a **seed,** or starting value, and an **increment,** which is the number to add each time a new record is added. Most commonly, the seed and increment are both set to 1, meaning that each new row will be given an identity value that is 1 higher than the preceding row.

Another option for an arbitrary PK is a GUID. GUIDs are most often used as PKs when you need to copy data between databases and you need to be sure that data copied from another database does not conflict with existing data. If you were instead to use identities, you would have to play with the seed values to avoid conflicts; for example, the number 1,000,454 could easily have been used in two databases, creating a conflict when the data is copied. The disadvantages of GUIDs are that they are larger than integers and they are not easily readable for humans. Also, PKs are often clustered, meaning that they are stored in order. Because GUIDs are random, each time you add data it ends up getting inserted into the middle of the PK, and this adds overhead to the operation. In Chapter 10 we talk more about clustered versus nonclustered PKs.

Of all the PK options we have discussed, we most often use identity columns. They are easy to set up and they provide consistency across tables. No matter what method you use, carefully consider the pros and cons. Implementing a PK in the wrong way not only will make it difficult to write code against your database but also could lead to degraded performance.

Foreign Keys

As with primary keys, foreign keys in SQL Server work in the same way as they do in logical design. A foreign key is the column or columns that correspond to a primary key and establish a relationship. Exactly the same columns with the same data as the primary key exist in the foreign key. It is for this reason that we strongly advise against using composite primary keys; not only does it mean a lot of data duplication, but also it adds overhead when you join tables. Going back to our employee and vehicle example, take a look at Figure 3.2, which shows the tables with some sample data.

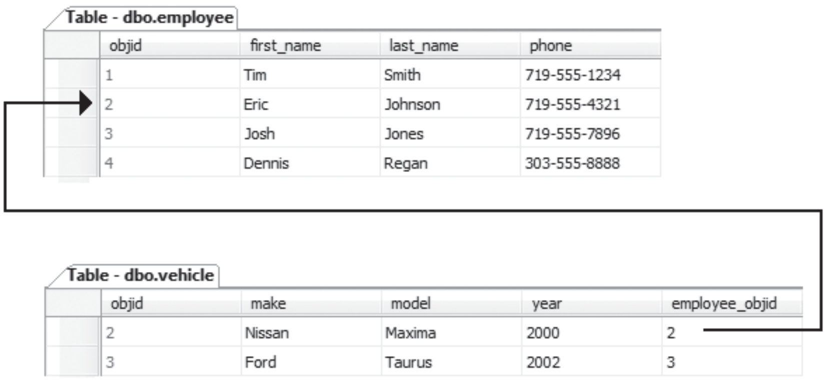


Table - dbo.employee

objid	first_name	last_name	phone
1	Tim	Smith	719-555-1234
2	Eric	Johnson	719-555-4321
3	Josh	Jones	719-555-7896
4	Dennis	Regan	303-555-8888

Table - dbo.vehicle

objid	make	model	year	employee_objid
2	Nissan	Maxima	2000	2
3	Ford	Taurus	2002	3

FIGURE 3.2 Data from the employee and vehicle tables showing the relationship between the tables

As you can see, both tables have objid columns. These are identity columns and serve as our primary key. Additionally, notice that the vehicle table has an employee_objid column. This column holds the objid of the employee to whom the car is assigned. In SQL Server, the foreign key is set up on the vehicle table, and its job is to ensure that the value you enter in the employee_objid column is in fact a valid value that has a corresponding record in the employee table.

The following script creates the vehicle table. You will notice a few things that are different from the earlier table creation script. First, when we set up the objid column, we use the `IDENTITY(1,1)` statement to create an identity, with a seed and increment of 1 on the column. Second, we have a second `CONSTRAINT` statement to add the foreign key relationship. When creating a foreign key, you specify the column or columns in the referencing table that contain the foreign key as well as the referenced table and columns that contain the primary key.

```
CREATE TABLE dbo.vehicle(
    objid int IDENTITY(1,1) NOT NULL,
    make varchar(50) NOT NULL,
    model varchar(50)NOT NULL,
    year char(4) NOT NULL,
    employee_objid int NOT NULL,
 CONSTRAINT PK_vehicle PRIMARY KEY (objid),
 CONSTRAINT FK_vehicle_employee
    FOREIGN KEY(employee_objid)
    REFERENCES employee (objid)
)
```

Once your primary keys are in place, the creation of the foreign keys is academic. You simply create the appropriate columns on the referencing table and add the foreign key. As stated in Chapter 2, if your design requires it, the same column in a table can be in both the primary key and a foreign key.

When you create foreign keys, you can also specify what to do if an update or delete is issued on the parent table. By default, if you attempt to delete a record in the parent table, the delete will fail because it would result in orphaned rows in the referencing table. An **orphaned row** is a row that exists in a child table that has no corresponding parent. This can cause problems in some data models. In our employee and vehicle tables, a NULL in the vehicle table means that the vehicle has not been assigned to an employee. However, consider a table that stores orders and order details; in this case, an orphaned record in the order detail table would be useless. You would have no idea which order the detail line belonged to.

Instead of allowing a delete to fail, you have options. First, you can have the delete operation **cascade,** meaning that SQL Server will delete all the child rows along with the parent row you are deleting. Be very careful when using this option. If you have several levels of relationships with cascading delete enabled, you could wipe out a large chunk of data by issuing a delete on a single record.

Your second option is to have SQL Server set the foreign key column to NULL in the referencing table. This option creates orphaned records, as discussed. Third, you can have SQL Server set the foreign key column back to the default value of the column, if it has one. Similar options are also available if you try to update the primary key value itself. Again, SQL Server can either (1) cascade the update so that the child rows still point to the correct parent rows with the new key, (2) set the foreign key to NULL, or (3) set the foreign key back to its default value.

Changing the values of primary keys isn't something we recommend you do often, but in some situations you may find yourself needing to do just that. If you find yourself in that situation often, you might consider setting up an update rule on your foreign keys.

Constraints

SQL Server contains several types of constraints to enforce data integrity. **Constraints,** as the name implies, are used to constrain the values that can be entered into columns. We have talked about two of the constraints in SQL Server: primary keys and foreign keys. Primary keys constrain the data so that duplicates and NULLs cannot exist in the columns, and foreign keys ensure that the entered value exists in the referenced table. There are several other constraints you can implement to ensure data integrity or enforce business rules.

Unique Constraints

Unique constraints are similar to primary keys; they ensure that no duplicates exist in a column or collection of columns. They are configured on columns that do not participate in the primary key. How does a unique constraint differ from a primary key? From a technical standpoint, the only difference is that a unique constraint allows you to enter NULL values; however, because the values must be unique, you can enter only one NULL value for the entire column. When we talked about identifying primary keys, we talked about candidate keys. Because candidate keys should also be able to uniquely identify the row, you should probably place unique constraints on your candidate keys. You add a unique constraint in much the same way as you add a foreign key, using a constraint statement such as

```
CONSTRAINT UNQ_vehicle_vin UNIQUE NONCLUSTERED (vin_number)
```

Check Constraints

Check constraints limit the values that can be entered into a column by using a logical expression. A **logical expression** is any SQL expression that can evaluate to TRUE or FALSE. The expression can be any valid SQL expression, from simple comparisons to something more complex such as calling a function. For example, say we want to limit the values that can be entered for salary in our employee table. The expression we would use to evaluate the data would be something like this:

```
salary >= 10000 and salary <=150000
```

This line rejects any value less than 10,000 or greater than 150,000.

Each column can have multiple check constraints, or you can reference multiple columns with a single check. When it comes to NULL values, check constraints can be overridden. When a check constraint does its evaluation, it allows any value that does not evaluate to false. This means that if your check evaluates to NULL, the value will be accepted. Thus, if you enter NULL into the salary column, the check constraint returns unknown and the value is inserted. This feature is by design, but it can lead to unexpected results, so we want you to be aware of this.

Check constraints are created in much the same way as keys or unique constraints; the only caveat is that they tend to contain a bit more meat. That is, the expression used to evaluate the check can be lengthy and therefore hard to read when viewed in T-SQL. We recommend you create your tables first and then issue `ALTER` statements to add your check constraints. The following sample code adds a constraint to the Products table to ensure that certain columns do not contain negative values.

```
ALTER TABLE dbo.Products
ADD CONSTRAINT chk_non_negative_values
CHECK
(
weight >= 0
AND (shippingweight >= 0 AND shippingweight >= weight)
AND height >= 0
AND width >= 0
AND depth >= 0
)
```

Because it doesn't make sense for any of these columns to contain negative numbers (items cannot have negative weights or heights), we add this constraint to ensure data integrity. Now when you attempt to insert data with negative numbers, SQL Server simply returns the following error and the insert is denied. This constraint also prevents a shipping weight from being less than the product's actual weight.

```
The INSERT statement conflicted with the CHECK constraint
"chk_non_negative_values"
```

As you can see, we created one constraint that looks at all the columns that must contain non-negative values. The only downfall to this method is that it can be hard to find the data that violated the constraint. In this case, it's pretty easy to spot a negative number, but imagine if the constraint were more complex and contained more columns. You would know only that some column in the constraint was in violation, and you would have to go over your data to find the problem. On the other hand, we could have created a constraint for each column, making it easier to track down problems. Which method you use depends on complexity and personal preference.

Implementing Referential Integrity

Now that we have covered PKs, FKs, and constraints, the final thing we need to discuss is how to use them to implement referential integrity. Luckily it's straightforward once you understand how to create each of the objects we've discussed.

One-to-Many Relationships

One-to-many relationships are the most common kind of relationship you will use in a database, and they are also what you get with very little additional work when you create a foreign key on a table. To make the relationship required, you must make sure that the column that contains your foreign key is set to not allow NULLs. Not allowing NULLs requires that a value be entered in the column, and adding the foreign key requires that the value be in the related table's primary key. This type of relationship implements a cardinality of "one or more to one." In other words, you can have a single row but you are not limited to the total number of rows you can have. (Later in this chapter we look at ways to implement advanced cardinality.) Allowing NULL in the foreign key column makes the relationship optional—that is, the data is not required to be related to the reference table. If you were tracking computers in a table and using a relationship to define which person was using the computer, a NULL in your foreign key would denote a computer that is not in use by an employee.

One-to-One Relationships

One-to-one relationships are implemented in exactly the same way as one-to-many relationships—sort of. You still create a primary key and a foreign key; the problem is that at this point SQL Server still allows users to insert many rows into the foreign key table that reference the primary key table.

There is no way, by default, to constrain the data to one-to-one. To implement a one-to-one relationship that is enforced, you must get a little creative.

The first option is to write a stored procedure (more on stored procedures later in this chapter) to do all your inserting, and then add logic to prevent a second row from being added to the table. This method works in most cases, but what if you need to load data directly to tables without a stored procedure? Another option to implement one-to-one relationships is to use a trigger, which we also look at shortly. Basically, a **trigger** is a piece of code that can be executed after or instead of the actual insert statement. Using this method, you could roll back any insert that would violate the one-to-one relationship.

Additionally—and this is probably the easiest method—you can add a unique constraint on the foreign key columns. This would mean that the data in the foreign key would have to be a value from the primary key, and each value could appear only once in the referencing table. This approach effectively creates a one-to-one relationship that is managed and enforced by SQL Server.

Many-to-Many Relationships

One of the most complex relationships when it comes to implementation is the many-to-many relationship. Even though you can have a many-to-many relationship between two entities, you cannot create a many-to-many relationship between only two tables. To implement this relationship, you must create a third table, called a **junction table,** and two one-to-many relationships.

Let's walk through an example to see how it works. You have two tables—one called Student and one called Class—and both contain an identity called objid as their PK. In this situation you need a many-to-many relationship, because each student can be in more than one class and each class will have more than one student. To implement the relationship, you create a junction table that has only two columns: one containing the student_objid, and the other containing the class_objid. You then create a one-to-many relationship from this junction table to the Student table, and another to the Class table. Figure 3.3 shows how this relationship looks.

You will notice a few things about this configuration. First, in addition to being foreign keys, these columns are used together as the primary key for the Student_Class junction table. How does this implement a many-to-many relationship? The junction table can contain rows as long as they do

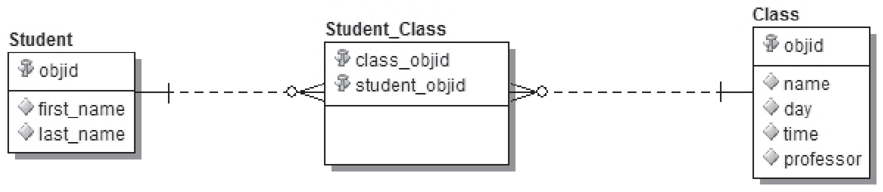


Figure 3.3 Many-to-many relationship between a Student and a Class table

not violate the primary key. This means that you can relate each student to all the classes he attends, and you can relate all the students in a particular class to that class. This gives you a many-to-many relationship.

It may sound complex, but once you create a many-to-many relationship and add some data to the tables, it becomes pretty clear. The best way to really understand it is to do it. When we build our physical model in Chapter 9, we look more closely at many-to-many relationships, including ways to make them most useful.

Implementing Advanced Cardinality

In Chapter 2, we talk about cardinality. Cardinality simply describes the number of rows in a table that can relate to rows in another table. Cardinality is often derived from your customer's business rules. As with one-to-one relationships, SQL Server does not have a native method to support advanced cardinality. Using primary and foreign keys, you can easily enforce one-or-more-to-many, zero-or-more-to-many, or one-to-one cardinality as we have described previously.

What if you want to create a relationship whereby each parent can contain only a limited number of child records? For example, using our employee and vehicle tables, you might want to limit your data so that each employee can have no more than five cars assigned. Additionally, employees are not required to have a car at all. The cardinality of this relationship is said to be zero-to-five-to-many. To enforce this requirement, you need to be creative. In this scenario you could use a trigger that counts the number of cars assigned to an employee. If the additional car would put the employee over five, the insert could be reversed or rolled back.

Each situation is unique. In some cases you might be able to use check constraints or another combination of PKs, FKs, and constraints to implement your cardinality. You need to examine your requirements closely to decide on the best approach.

Programming

In addition to the objects that are used to store data and implement data integrity, SQL Server provides several objects that allow you to write code to manipulate your data. These objects can be used to insert, update, delete, or read data stored in your database, or to implement business rules and advanced data integrity. You can even build "applications" completely contained in SQL Server. Typically, these applications are very small and usually manipulate the data in some way to serve a function or for some larger application.

Stored Procedures

Most commonly, when working with code in SQL Server you will work with a **stored procedure** (SP). SPs are simply compiled and stored T-SQL code. SPs are similar to views in that they are compiled and they generate an execution plan when called the first time. The difference is that SPs, in addition to selecting data, can execute any T-SQL code and can work with parameters. SPs are very similar to modules in other programming languages. You can call a procedure and allow it to perform its operation, or you can pass parameters and get return parameters from the SP.

Like columns, **parameters** are configured to allow a specific data type. All the same data types are used for parameters, and they limit the kind of data you can pass to SPs. Parameters come in two types: input and output. **Input parameters** provide data to the SP to use during their execution, and **output parameters** return data to the calling process. In addition to retrieving data, output parameters can be used to provide data to SPs. You might do this when an SP is designed to take employee data and update a record if the employee exists or insert a new record if the employee does not exist. In this case, you might have an EmployeeID parameter that maps to the employee primary key. This parameter would accept the ID of the employee you intend to update as well as return the new employee ID that is generated when you insert a new employee.

SPs also have a return value that can return an integer to the calling process. **Return values** are often used to give the calling process information about the success of the stored procedure. Return values differ from output parameters in that return values do not have names and you get only one per SP. Additionally, SPs always return an integer in the return value, even if you don't specify that one be returned. By default, an SP returns 0 (zero) unless you specify something else. For this reason, 0 is often used to designate success and nonzero values specify return error conditions.

SPs have many uses; the most common is to manage the input and retrieval of your data. Often SPs are mapped to the entities you are storing. If you have student data in your database, you may well have SPs named sp_add_student, sp_update_student, and sp_retrieve_student_data. These SPs would have parameters allowing you to specify all the student data that ultimately needs to be written to your tables.

Like views, SPs reduce your database's complexity for users and are more efficient than simply running T-SQL repeatedly. Again, SPs remove the need to update application code if you need to change your database. As long as the SP accepts the same parameters and returns the same data after you make changes, your application code does not have to change. In Chapter 11 we talk in great detail about using stored procedures.

User-Defined Functions

Like any programming language, T-SQL offers functions in the form of **user-defined functions** (UDFs). UDFs take input parameters, perform an action, and return the results to the calling process. Sound similar to a stored procedure? They are, but there are some important differences. The first thing you will notice is a difference in the way UDFs are called. Take a look at the following code for calling an SP.

```
DECLARE @num_in_stock int

EXEC sp_check_product_stock    @sku = 4587353,
          @stock_level = @num_in_stock OUTPUT

PRINT @num_in_stock
```

You will notice a few things here. First, you must declare a variable to store the return of the stored procedure. If you want to use this value later, you need to use the variable; that's pretty simple.

Now let's look at calling a UDF that returns the same information.

```
DECLARE @num_in_stock int

SET @num_in_stock = dbo.CheckProductStock (4587353)

PRINT @num_in_stock
```

The code looks similar, but the function is called more like a function call in other programming languages. You are probably still asking yourself, "What's the difference?" Well, in addition to calling a function and putting its return into a variable, you can call UDFs inline with other code. Consider the following example of a UDF that returns a new employee ID. This function is being called inline with the insert statement for the employee table. Calling UDFs in this way prevents you from writing extra code to store a return variable for later use.

```
INSERT INTO employee (employeeid, firstname, lastname)
VALUES (dbo.GetNewEmployeeID(), 'Eric', 'Johnson')
```

The next big difference in UDFs is the type of data they return. UDFs that can return single values are known as **scalar functions.** The data the function returns can be defined as any data type except for text, ntext, image, and timestamp. To this point, all the examples we have looked at have been scalar values.

UDFs can also be defined as **table-valued functions:** functions that return a table data type. Again, table-valued functions can be called inline with other T-SQL code and can be treated just like tables. Using the following code, we can pass the employee ID into the function and treat the return as a table.

```
SELECT * FROM dbo.EmployeeData(8765448)
```

You can also use table-valued functions in joins with other functions or with base tables. UDFs are used primarily by developers who write T-SQL code against your database, but you can use UDFs to implement business rules in your model. UDFs also can be used in check constraints or triggers to help you maintain data integrity.

Triggers

Triggers and constraints are the two most common ways to enforce data integrity and business rules in your physical database. Triggers are stored T-SQL scripts, similar to stored procedures, that run when a DML statement (other than SELECT) is issued against a table or view. There are two types of DML triggers available in SQL Server.

With an **AFTER trigger,** which can exist only on tables, the DML statement is processed, and after that operation completes, the trigger code is run. For example, if a process issues an insert to add a new employee to a table, the insert triggers the trigger. The code in the trigger is run after the insert as part of the same transaction that issued the insert. Managing transactions is a bit beyond the scope of this book, but you should know that because the trigger is run in the same context as the DML statement, you can make changes to the affected data, up to and including rolling back the statement. AFTER triggers are very useful for verifying business rules and then canceling the modification if the business rule is not met.

During the execution of an AFTER trigger, you have access to two virtual tables—one called Inserted and one called Deleted. The Deleted table holds a copy of the modified row or rows as they existed before a delete or update statement. The Inserted table has the same data as the base table has after an insert or update. This arrangement allows you to modify data in the base table while still having a reference to the data as it looked before and after the DML statement.

These special temporary tables are available only during the execution of the trigger code and only by the trigger's process. When creating AFTER triggers, you can have a single trigger fire on any combination of insert, update, or delete. In other words, one trigger can be set up to run on both insert and update, and a different trigger could be configured to run on delete. Additionally, you can have multiple triggers fire on the same statement; for example, two triggers can run on an update. If you have multiple triggers for a single statement type, the ordering of such triggers is limited. Using a system stored procedure, sp_settriggerorder, you can specify which trigger fires first and which trigger fires last. Otherwise, they are fired in the middle somewhere. In reality, this isn't a big problem. We have seen very few tables that had more than two triggers for any given DML statement.

INSTEAD OF triggers are a whole different animal. These triggers perform in the way you would expect: The code in an INSTEAD OF trigger fires in place of the DML statement that caused the trigger to fire. Unlike AFTER triggers, INSTEAD OF triggers can be defined on views as well as tables. Using them, you can overcome the limitation of views that have multiple base tables. As mentioned earlier, you can update a view only if you limit your update to affecting only a single base table. Using an INSTEAD OF trigger, you can update all the columns of a view and use the trigger to issue the appropriate update against the appropriate base table. You can also use INSTEAD OF triggers to implement advanced data integrity or business rules by completely changing the action of a DML statement.

You can also control trigger nesting and recursion behavior. With nested triggers turned on, one trigger firing can perform a DML and cause another trigger to fire. For example, inserting a row into TableA causes TableA's insert trigger to fire. TableA's insert trigger in turn updates a record in TableB, causing TableB's update trigger to fire. That is **trigger nesting**—one trigger causing another to fire—and this is the default behavior. With nested triggers turned on, SQL Server allows as many as 32 triggers to be nested. The INSTEAD OF trigger can nest regardless of the setting of the nested triggers option.

Server trigger recursion specifies whether or not a trigger can perform a DML statement that would cause the same trigger to fire again. For example, an update trigger on TableA issues an additional update on TableA. With recursive triggers turned on, it causes the same trigger to fire again. This setting affects only direct recursion; that is, a trigger directly causes itself to fire again. Even with recursion off, a trigger could cause another trigger to fire, which in turn could cause the original trigger to fire again. Be very careful when you use recursive triggers. They can run over and over again, causing a performance hit to your server.

CLR Integration

As of SQL Server 2005, we gained the ability to integrate with the .NET Framework Common Language Runtime (CLR). Simply put, CLR integration allows you to use .NET programming languages within SQL Server objects. You can create stored procedures, user-defined functions, triggers, and CLR user-defined types using the more advanced languages available in Microsoft .NET. This level of programming is beyond the scope of this book, but you need to be aware of SQL Server's ability to use CLR. You will likely run into developers who want to use CLR, or you may find yourself needing to implement a complex business rule that cannot easily be implemented using standard SQL Server objects and T-SQL. So if you are code savvy or have a code-savvy friend, you can create functions using CLR to enforce complex rules.

Implementing Supertypes and Subtypes

We discuss supertypes and subtypes in Chapter 2. These are entities that have several kinds of real-world objects being modeled. For example, we might have a supertype called phone with subtypes for corded and cordless phones. We separate objects into a subtype cluster because even though a phone is a phone, different types will require that we track different attributes. For example, on a cordless phone, you need to know the working range of the handset and the frequency on which it operates, and with a corded phone, you could track something like cord length. These differences are tracked in the subtypes, and all the common attributes of phones are held in the supertype.

How do you go about physically implementing a subtype cluster in SQL Server? You have three options. The first is to create a single table that represents the attributes of the supertype and also contains the attributes of *all* the subtypes. Your second option is to create tables for each of the subtypes, adding the supertype attributes to each of these subtype tables. Third, you can create the supertype table and the subtype tables, effectively implementing the subtype cluster in the same way it was logically modeled.

To determine which method is correct, you must look closely at the data being stored. We will walk through each of these options and look at the reasons you would use them, along with the pros and cons of each.

Supertype Table

You would choose this option when the subtypes contain few or no differences from the data stored in the supertype. For example, let's look at a cluster that stores employee data. While building a model, you discover that the company has salaried as well as hourly employees, and you decide to model this difference using subtypes and supertypes. After hashing out all the requirements, you determine that the only real difference between these types is that you store the annual salary for the salaried employees and you need to store the hourly rate and the number of hours for an hourly employee.

In this example, the subtypes contain very subtle differences, so you could build this subtype cluster by using only the supertype table. For this situation, you would likely create a single employee table that contains all the attributes for employees, including all three of the subtype attributes for salary, hourly rate, and hours. Whenever you insert an hourly employee, you would require that data be in the hourly rate and hour columns and that the salary column be left NULL. For salaried employees, you would do the exact opposite.

Implementing the types in this way makes it easy to find the employee data because all of it is in the same place. The only drawback is that you must implement some logic to look at the columns that are appropriate to the type of employee you are working with. This supertype-only implementation works well only because there are very few additional attributes from the subtype's entities. If there were a lot of differences, you would end up with many of the columns being NULL for any given row, and it would take a great deal of logic to pull the data together in a meaningful way.

Subtype Tables

When the data contained in the subtypes is dissimilar and the number of common attributes from the supertype is small, you would most likely implement the subtype tables by themselves. This is effectively the opposite data layout that would prompt you to use the supertype-only model.

Suppose you're creating a system for a retail store that sells camera equipment. You could build a subtype cluster for the products that the store sells, because the products fall into distinct categories. If you look only at cameras, lenses, and tripods, you have three very different types of product. For each one, you need to store the model number, stock number, and the product's availability, but that is where the similarities end. For cameras you need to know the maximum shutter speed, frames per second, viewfinder size, battery type, and so on. Lenses have a different set of attributes, such as the focal length, focus type, minimum distance to subject, and minimum aperture. And tripods offer a new host of data; you need to store the minimum and maximum height, the planes on which it can pivot, and the type of head. Anyone who has ever bought photography equipment knows that the differences listed here barely scratch the surface; you would need many other attributes on each type to accurately describe all the options.

The sheer number of attributes that are unique for each subtype, and the fact that they have only a few in common, will push you toward implementing only the subtype tables. When you do this, each subtype table will end up storing the common data on its own. In other words, the camera, lens, and tripod tables would have columns to store model numbers, SKU numbers, and availability. When you're querying for data implemented in this way, the logic needs to support looking at the appropriate table for the type of product you need to find.

Supertype and Subtype Tables

You have probably guessed this: When there are a good number of shared attributes and a good number of differences in the subtypes, you will probably implement both the supertype and the subtype tables. A good example is a subtype cluster that stores payment information for your customers. Whether your customer pays with an electronic check, credit card, gift certificate, or cash, you need to know a few things. For any payment, you need to know who made it, the time the payment was received, the amount, and the status of the payment. But each of these payment types also requires you to know the details of the payment. For credit cards, you need the card number, card type, security code, and expiration date. For an electronic check, you need the bank account number, routing number, check number, and maybe even a driver's license number. Gift cards are simple; you need only the card number and the balance. As for cash, you probably don't need to store any additional data.

This situation calls for implementing both the supertype and the subtype tables. A Payment table could contain all the high-level detail, and individually credit card, gift card, and check tables would hold the information pertinent to each payment type. We do not have a cash table, because we do not need to store any additional data on cash payments beyond what we have in the Payment table.

When implementing a subtype cluster in this way, you also need to store the subtype **discrimination,** usually a short code or a number that is stored as a column in the supertype table to designate the appropriate subtype table. We recommend using a single character when possible, because they are small and offer more meaning to a person than a number does. In this example, you would store CC for credit card, G for a gift card, E for electronic check, and C for cash. (Notice that we used CC for a credit card to distinguish it from cash.) When querying a payment, you can join to the appropriate payment type based on this discriminator.

If you need data only from either the supertype or the subtype, this method offers two benefits: you need go to only one table, and you don't retrieve extraneous data. However, the flip side is that you must determine which subtype table you need to query and then join both tables if you need data from both the supertype and a subtype table. Additionally, you may find yourself needing information from the supertype and multiple subtypes; this will add overhead to your queries because you must join multiple tables.

Supertypes and Subtypes: A Final Word

Implementing supertypes and subtypes can, at times, be tricky. If you take the time to fully understand the data and look at the implications of splitting the data into multiple tables versus keeping it tighter, you should be able to determine the best course of action. Don't be afraid to generate some test data and run various options through performance tests to make sure you make the correct choice. When we get to building the physical model, we look at using subtype clusters as well as other alternatives for especially complex situations.

Summary

In this chapter, we have looked at the available objects inside SQL Server that you will use when implementing your physical model. It's important to understand these objects for many reasons. You must keep all this in mind when you design your logical model so that you design with SQL Server in mind. This also plays a large part later when you build and implement your physical model. You will probably not use every object in SQL Server for every database you build, but you need to know your options. Later, we walk through creating your physical model, and at that time we go over the various ways you can use these physical objects to solve problems.

In the next chapter, we talk about normalization, and then we move on to the meat and potatoes of this book by getting into our sample project and digging into a lot of real-world issues.

This page intentionally left blank