

Relational Databases 101

Introduction

Many of my readers come from backgrounds that don't include formal training on the best ways to design and create efficient, business-class *relational* databases. If you arrive here with Microsoft Access or FoxPro experience, you're at an advantage—you know that, for the most part, the process of creating a database is hidden from you by the application's IDE—you just use drag-and-drop or use wizards to build the databases and tables you want. That's not at all bad, but without an in-depth understanding of how to best create, tune, and protect a *relational* database, I suspect that the relational “normalcy,” relational integrity constraints, performance, and scalability of the result might not be particularly stellar. And, more importantly, the data might not be particularly secure. By “normalcy,” I mean how well the database conforms to the recognized standards of relational design where database designers attempt to “normalize” databases to at least the third level. If you're not sure how to do this, or even what this means, have no fear—I'll explain this later. The SQL gurus with whom I work (like Peter Blackburn, Kimberly Tripp, and a litany of others) are convinced that more problems can be solved by efficient database design than by the cleverest, best-written application front-end.

IMHO It's not how fast you ask the question—it's how long it takes to find the answer that gates performance.

Getting Started with Solid Database Design

The Microsoft Books Online (BOL) documentation seems to fall a bit short in this important subject, so this chapter might be helpful for those who need a more complete understanding of how to create a best-practice relational database. The problem faced by any database designer is knowing what's going to be stored ahead of time—before the first table is created. That's always been (and always will be) a problem. As I've said before, a customer rarely knows what they want until they don't get it.

To get started on the right foot, I recommend a good course in relational theory like *Extended Relational Analysis*. This can do a world of good—but its depth is well beyond the scope of this book. In courses like this, you learn how to ask the right questions for each “entity” you expect to store in the database.

I also think that using a (big) whiteboard to lay out the database with your team (or customer) can help visualize the data. Getting everyone who is going to consume the data is essential. How I design a database for a single-focused project is very different than the way it's designed for projects where a small multitude of groups expect to consume the data. Admittedly, database development by committee is tough, and one should try to avoid those situations, but leaving town might not be an option.

Before we get into the academics of normalization, let's spend a few moments in quiet contemplation and focus on a few guiding principles. As you design your database, you should keep these basic tenants in mind:

- Each table needs a unique identifier. That is, you need to choose one or more columns to permit SQL Server to find specific rows to return or update without including other irrelevant rows. In SQL Server, this typically means each table should have an “ID” column (typically cast as an *Identity integer*) that gives the row a unique (SQL Server-generated) value. You should define this column using the Primary Key (PK) constraint (as discussed in Chapter 2, “How Does SQL Server Work?”). For example *Au_ID* is the unique identifier for the *Authors* table in the sample *Biblio* database.
- Each table should store only one kind of information and not repeat information in multiple columns. As I discuss next, this is

where normalization comes in. SQL Server is tuned to work with small, tight rows that contain relatively few columns. If you find your table has more than a dozen columns, you're treading off the boards and into the swamp. Remember, the largest row you can define is only 8K (not counting BLOB columns).

- Microsoft feels that you should avoid columns that can be set to NULL—I'm not so sure. That is, they feel that you should avoid columns where you might not have access to the data at all and cannot (logically) assign an arbitrary default. Each time you define a column as permitting NULL (making it “nullable”), SQL Server incurs extra overhead. It makes sense to move these columns to another table and cross-reference them to the table's PK as long as the database complexity does not get out of hand.
- Each column needs to be defined both with the content in mind and with the constraints needed to keep it pure. It's not enough to type a column as *integer* and hope that the data entered therein is going to be pure. All columns, especially numeric columns, need to have (at least) CHECK constraints defined, if not TSQL rules. *Columns* should be defined to hold what's expected to be saved—and no more. Needlessly bloating data type capacity simply chews up memory to no good purpose. If you have columns that contain text, but that text is never expressed in Unicode, don't use a Unicode type. If you have a date column but don't need to store time with 3.33-millisecond accuracy, use *smalldatetime* instead of *datetime*. You get the idea. In this case, less is more—more space saved, more memory available to store other pertinent stuff. I'll show you how much each column costs in memory near the end of this chapter.

IMHO Understand that all data is evil until proven innocent—it's not in the U.S. Constitution, but given the state of the current Congress, it just might get there.

- Start thinking about a concurrency strategy from the beginning. Determine how data is to be shared (if at all). Consider that many “single-user” databases are doomed to failure once they are “converted” to multi-user. Think about the mechanism you're planning to use to determine if a row has changed once it's fetched.

For example, you might (perhaps ought) to use a “*Rowversion*” or “*TimeStamp*” column to help track access—it can make Visual Studio’s job a lot easier (and yours, too) when it comes time to write action commands to change the database.

- Avoid BLOBs in the database. I have been suggesting this for over a decade, and those who have listened have been able to build a smaller, faster, and simpler database. If you have BLOBs, store them in files (or on RO media) and use the database to store the path.
- Finally, and perhaps most importantly, for less-experienced developers, I think that you should strive to keep your database simple—simple to understand, support, and maintain. Excessive complexity is the bane of many a mature and amateur database designer.

Understanding Relational Database Normalization

In a nutshell, building a “good” optimized, relational database is mostly about *normalization*. Once you understand the basic principles of normalization, SQL Server should be able to manage your data more efficiently, the applications you write should be able to return data more efficiently, and you’ll find it a lot easier to protect your database’s data and relational integrity.

So, what is “normalization” and how does it help performance and all that other good stuff? Well, normalization is simply the set of relational database techniques developed to efficiently organize the information you want to manage in a relational database. The academics talk about (at least) five “normal forms,” but most database designers stick to the first three forms and seldom go further. The benefit of implementing further levels is usually not that great when compared to the costs—especially in smaller databases.

Here are basic tenants of the first three normal forms.

1. **First normal:** Don’t define duplicate columns in the same table. Each column in a table should contain “different” information. This does not mean avoiding use of identical column names (that’s prohibited by the SQL engine), but it does mean

that any two columns should not store basically the same information. For example, don't create a table with two or more addresses for a customer (such as a home and business address), as shown in Figure 3.1. The solution to this problem is best implemented by the second normal form.

Column Name	Data Type	Allow Nulls
CustID	int	<input type="checkbox"/>
CustName	nvarchar(50)	<input type="checkbox"/>
HomeAddress	nvarchar(50)	<input checked="" type="checkbox"/>
OfficeAddress	nvarchar(50)	<input checked="" type="checkbox"/>
VacationAddress	nvarchar(50)	<input checked="" type="checkbox"/>
RowVersion	timestamp	<input type="checkbox"/>

Figure 3.1 Unnormalized Customers table.

- 2. Second normal:** All attributes (columns) in a table that are not dependent on the primary key must be eliminated. This means you need to create a separate table for each logical group of data and identify each row with a unique set of columns (its own primary key). In this case, create a separate *Addresses* table and connect the two tables together with their primary keys, as shown in Figure 3.2.
- 3. Third normal:** Tables cannot include duplicate information. For example, if two tables require a common field, a separate table should be created to manage that column. Our basic design already conforms to the third normal form. However, as an example, take a look at the Biblio database. In this case, I created a *TitleAuthor* table that contains fields common to the *Titles* and *Authors* table.

No, this is not an in-depth discussion of normalization or relational theory, but it's enough to get you started. It's also important to know that many database developers bend these rules from time to time to get more efficiency out of their databases. Sometimes, they add a bit more detail for an entity in a parent table, so it's not always necessary to JOIN to another table just to get one or two bits of information. Yes, these changes mean that the data must be kept current in two different tables, and if someone else comes along and does not realize what's going on....

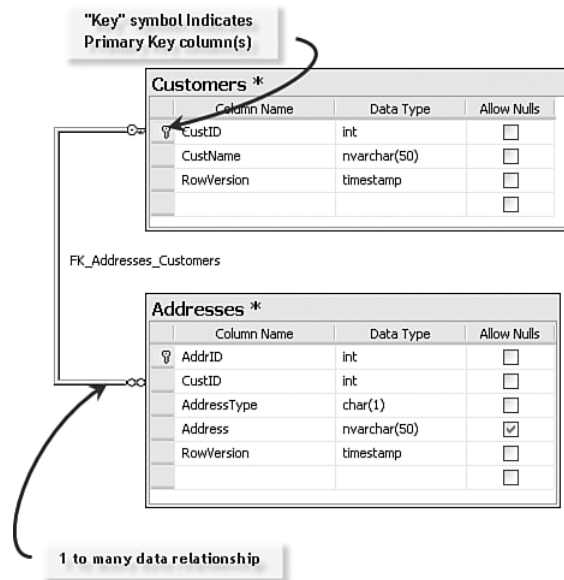


Figure 3.2 Normalized Customers and Addresses tables.

Understand as well that stored procedures or object-based approaches can (and do) help resolve these issues. By blocking direct access to base tables, developers can write server-side code to dereference the data in the base tables and get away with some tactics that would cause quite a bit of trouble if direct table access were permitted.

Once you have decided what tables you need, you need to use one of the SQL Server or Visual Studio tools to create them (as I illustrate in Chapter 4, “Getting Started with Visual Studio”). But before doing that, I often draw these tables on a whiteboard, which makes it easier to “see” how the data is to be stored and how the tables are related. In Visual Studio, you can use the database diagramming tool to help at this phase, and the ink does not stain your fingers as much.

Creating Tables, Rows, and Columns

Relational databases are defined in fairly simple terms¹:

- **Database:** An extensible collection of related data typically organized as a set of tables. For example, an accounting database would contain information about the customers, inventory, orders, items, and other details of the accounting operation. Where and how the data is stored, protected, fetched, and updated is irrelevant, as it is managed entirely by the database management system (DBMS). The key word here is “extensible.” Due to its fundamental design, a relational database is easily expanded to encompass more data entities to store.
- **Tables:** An extensible collection of rows containing related data. For example, the *Customers* table would contain a collection of rows pertaining to (just) customers—not the things they order, or sell, or where they bank—just about the individual customer.
- **Rows:** An extensible collection of column headings and typed columns to contain data details collected in a single table. Each row pertains to a single entity as a row in the *Customers* table would refer to a single customer. There is never an implied row order in a relational data table—this means, unless specifically requested, rows are returned in a nondeterministic order. This is especially true of SQL Server 2005 queries—with parallel processing, even rows stored with a clustered index can appear in any order.
- **Columns:** A named storage place for base-typed, user-defined typed, or (in the case of SQL Server) `sql_variant` “morphing” typed data. Columns are always returned in the order in which they are defined unless otherwise requested.

IMHO No, an ADO.NET *DataTable* or *TableAdapter* object is not synonymous with a database table.

¹ See C. J. Date, *An Introduction to Database Systems* (Volume 1, 4th Edition) (Addison-Wesley, 1986), p. 117.

How SQL Server Stores Relational Databases

SQL Server has expanded the number and type of objects managed and contained in the database to include collections of other objects such as logins, roles, users, stored procedures, views, triggers, functions, user-defined types, reports, and other objects; and in SQL Server 2005, assemblies, functions, aggregates, and CLR-based user-defined types (UDTs). In SQL Server, the definition of a column is expanded to include the ability to define columns whose datatype morphs to the datatype of the data stored on a row-by-row basis (`sql_variant`) or is defined by a CLR-based user-defined type.

SQL Server databases can contain billions of tables; tables have zero to virtually any number of rows, and rows contain 1 to 1,024 columns² but are (generally) limited in size to 8K³ (not counting BLOB and variable-length columns)⁴. But, no, I don't expect your database to have more than a few dozen to a few hundred tables. If you have more than a thousand tables, you have a *very* complex database. I guess SQL Server supports a virtually unlimited number of tables so Microsoft could say that SQL Server supports as many tables as Oracle or one of its other competitors. It's like saying your car can contain a billion marbles—just how many marbles does one car need to carry?

IMHO

If you find yourself working with a table that contains several hundred columns, there's usually something wrong with the design. Consider that the maximum size for a row (the sum of all data consumed by its columns) is about 8,000 bytes. Sure, some column data is stored on separate pages (like BLOBs and `varchar(max)` columns) and don't contribute (very much) to the total. Just make sure that your database is properly normalized before coming back to us when your rows are too large.

² In SQL Server 2005.

³ SQL Server 2005 supports row-overflow storage, which enables variable-length columns to be pushed off-row. Only a 24-byte root is stored in the main record for variable-length columns pushed out of row. Because of this, the effective row limit is higher than in previous releases of SQL Server. For more information, see the "Row-Overflow Data Exceeding 8KB" topic in SQL Server 2005 BOL.

⁴ See "Maximum Capacity Specifications for SQL Server 2005" in BOL.

Your data is ultimately stored in named and typed “columns.” The term “column” is synonymous with a “field” in an Index Sequential (ISAM) database like JET or a flat-file database. Okay, let’s go over those objects in a bit more detail.

Identifiers

The database, its “owner” (the user or schema that created the object), the table, and the columns are all referenced (addressed) using SQL Server identifier object names. These names can be up to 128 bytes in length, but I generally keep the names short. I don’t encourage anyone to embed spaces in the name, as it trips up the tools and your code—I also won’t support you if you do. Yes, you can name your column “Customer Last Name,” but you’ll need to surround this column name (or any object name that contains spaces) with square brackets: “[Customer Last Name].” Most of the tools do this anyway to protect themselves from folks that insist on using embedded spaces. I’m not nearly as tolerant. I prefer to separate these long names using the underscore (“_”) character or by using CamelCase, as in “CustomerLastName”.

When addressing a table in SQL Server and the server is named “Fred\SS1”, the database is “Biblio” and the schema⁵ is “Dev1”, you could address the “Sales” column in the “Customers” table by using the following identifier:

```
Fred1\SS1.Biblio.Dev1.Sales.Customers
```

Identifiers are case-sensitive only *if* you install your server in case-sensitive mode—I rarely do and I never encourage customers to do so. Installing an SQL Server as non-case-sensitive means you can define your columns using your company’s standard naming convention and not have to worry about the case.

No, you won’t be able to use special characters such as “[{}|;:”<, > . ! @ # \$ % ^ & * () + =” in any identifier. You’ll also discover that there is a long list of “reserved” keywords that can’t (should not) be used as object identifiers. This means you can’t call a database “Authorization”, name a column “Sort”, or name a Table “Select”.⁶ It also turns out that the ANSI SQL standards body has defined even *more* names that

⁵ I discuss the term “schema” later in this chapter.

⁶ See “Reserved Keywords” in Books Online.

are not yet reserved words in SQL Server. I would stay away from these, too. Actually, if you create compound names separated by an underscore (`_`) character, you should be safe with virtually any name. When I get to naming stored procedures a bit later, I'll also show why using `"sp_"` as a prefix for a stored procedure name is a bad idea—it forces the server to search for your stored procedure in the *master* database before looking in the current catalog.

Defining a Primary Key

When you define your table, you need to decide how to uniquely identify each row. No, this is not an absolute requirement, but it's unusual to have a table where each row cannot be located on its own. Ninety-nine percent of the business databases I've worked with over the years define one or more columns as the "primary key" (PK) for each table in the database. In some cases, there is no formally defined PK, but one could uniquely identify a row using one or more columns.

Using a person's name as the PK might be tempting, but as your database grows, there's an excellent chance that two or more people with the name "John Smith" will show up. Even when you're building a table for individuals, you might not want to (or might not be permitted to⁷) use the (U.S.) federal Social Security Account Number (SSAN) as a unique identifier. Frankly, I think it's a mistake to do so for a number of reasons. First, this is a very important piece of personal information that could mean an individual can have their identity stolen. Second, you need to consider that the SSAN is *not* a unique number. While the U.S. government does not (intentionally) assign duplicate SSANs, there are other nefarious individuals ("evil-doers") "issuing" SSANs to folks needing IDs to get jobs or credit. Third, SSANs are not given to everyone in the world—at least, not yet. Using a driver's license number is also not a good idea, for the same reasons. I expect that there will be a "DNA" ID before long that will help identify people—until someone shows up with a stolen thumb.

Using Identity or GUID Primary Keys

Virtually all of the databases I work with use a system-generated "identity" column or a globally unique identifier (GUID) (using the uniquei-

⁷ The Privacy Act of 1974 states: (Sec. 7(a) (1)) "It shall be unlawful for any Federal, State, or local government agency to deny to any individual any right, benefit, or privilege provided by law because of such individual's refusal to disclose his social security account number."

dentifier datatype) as the primary key in each table. For now, let's consider use of identity or GUID columns as the best choice for your primary key. What's the difference between the two? Well, the identity column is an integer that's generated for you by the server (and guaranteed to be unique in the scope of the table), and the GUID is a unique string that you ask the system to generate in code. It's also guaranteed to be unique, but globally (all over the world). Each of these primary keys has issues when it comes to using them in ADO.NET, as I discuss in Chapter 13, "Managing SQL Server CLR Executables." Unique identifiers also have an impact on your design as well. Consider these points:

- An identity column (*integer*) can be inspected, selected, or entered by your application's user far easier than a GUID. If you plan to let the user enter (I frown on this) or choose a PK from a list, you'll find that GUIDs are very hard to enter (correctly) and just as hard to pick from a list.
- If your data is spread out over a number of servers, the GUID is the best choice. That's because you can be guaranteed that the number generated in the remote site will be different from those generated locally or from other remote sites.
- GUIDs tend to spread out the index values more than integer identity values so data can be distributed more evenly across data pages. However, "sequential" retrieval is more expensive.
- The GUID is a far larger value, which is somewhat more expensive to store and manage in memory.
- You can use the NEWID TSQL function or .NET functions to set or initialize a GUID. Note that you won't get the same value twice.
- You can also create your own GUID string, as long as it's in the format (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx), in which each x is a hexadecimal digit in the range 0–9 or A–F). For example, 6F9619FF-8B86-D011-B42D-00C04FC964FF is a valid uniqueidentifier value.
- You can use the SCOPE_IDENTITY(), @@Identity, or other TSQL commands/functions to fetch the most recently created identity value to pass back to your client. I discuss these functions in Chapter 13 and show how to use them with an ADO.NET Update.

- Identity column values can also be set manually, and you can set the autoincrement and seed (starting) value in code. You can also use this technique to manage client-side identity values that need to tie parent and child tables together relationally but still be able to permit the server to generate “actual” identity values when you post data to the database. I also discuss this in detail in Chapter 13.

Setting Multi-Column Primary Keys

In more sophisticated databases, as you define your table, you’ll find it necessary to uniquely identify a row using *more* than one column. For example, suppose you’re working with a *Customers*, *Orders*, *Items* relational hierarchy of tables. In this case, there are many customers and each customer has zero or many orders, and each order has zero or many items. For this situation, I create three tables to store the information (as shown in Figure 3.3).

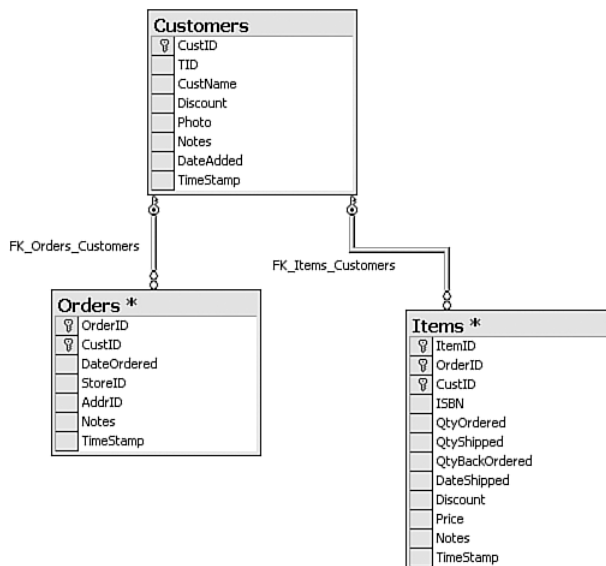


Figure 3.3 Defining multiple-column primary keys.

I set up *CustID* as the primary key (abbreviated PK) for the *Customers* table and set the datatype to *identity*. This uniquely identifies each customer with an SQL Server-generated integer value. The *OrderID* in the *Orders* table is another identity value, but I need the *CustID* to point to the customer that placed this order. These two

columns taken together form the PK for the *Orders* table. Likewise, the items associated with a specific order made by a specific customer are kept in three columns in the *Items* table. Using this strategy, I can locate the customer associated with a particular item without having to know the *OrderID*.

Understanding Parents and Children

Note that the database diagram shows the relationships among the three tables. In this case, there is a primary key/foreign key (PK/FK) relationship between the *Orders* and *Customers* table, as well as the *Items* and *Customers* table. A PK/FK relationship ties two tables together, in that when a row is added to the foreign key table, there is a corresponding row in the primary key table. This means you can't add an order with an invalid or missing customer ID (CustID). Because both of these tables (most tables) have a primary key, it can be bit confusing. For this reason (and other reasons), I call the "primary key" table the "parent" and the foreign key table the "child." In our design, the *Customer* table is the parent, and it has two children—the *Orders* and *Items* tables. I could also create a tiered parent/child hierarchy, as shown in Figure 3.4.

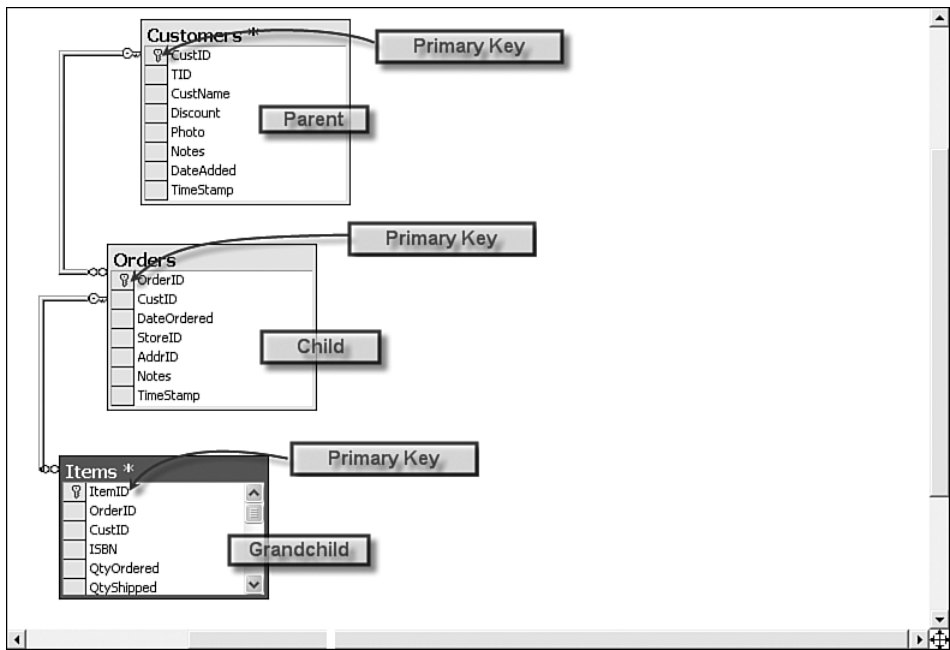


Figure 3.4 A parent/child relationship tree.

TIP These diagrams are annotated screenshots from a database diagram created by Visual Studio.

These relationships can be defined in the database to ensure that no order is created without a valid *CustID* and no item is created without a valid *OrderID* and a valid *CustID*. These defined (and server-enforced) relationships are called “constraints” and are used to maintain “referential” and data integrity. When these constraints are enabled, they mean that you won’t be able to delete customers from the database who have orders or items. When I start making changes in the database with ADO.NET, I’ll see how I have to handle these relationships with care. Note that once these relationships are defined in the database, no matter what applications access the database, these relationships are enforced. This means you can be (more) confident that when the pointy-haired manager starts to make changes to the data with Access, he (or she) won’t be able to break the referential integrity—or at least, not easily.

Changing the Primary Key

One other point before I move on. Once a primary key is created, it should be considered inviolate. If you think that a change to the primary key is necessary, think again. It’s far safer and easier to delete the current hierarchy and rebuild it rather than simply trying to change a primary key. If the constraints are in place (and you can disable them in code), the server won’t let you change the PK until all related dependencies are removed. That means you’ll need to delete all of the parent’s children (and all of the grandchildren) before changing or deleting the parent row. Since the parent might have a dozens of dependencies throughout the database, this is not an easy task.

Naming Objects

There are a few things to watch out for as you name databases, tables, columns, or any other object in the database. In TSQL jargon, object names are called “identifiers,” in case you want to look this up in BOL. These identifiers are created when the object is created and stored in the bowels of the master or user database. A handy place to find these names is the sysobjects table—if it still exists. The identifier specification breaks objects down into two groups: “regular” and “delimited” identifiers. The only real difference is that if the identifier does not comply with the rules for creating identifiers, it must be bracketed with double quotes or the

bracket ([]) symbols. For example, “This is a column name” and [This is another column name] are delimited identifiers.

- Object names must be unique in their scope. That is, database names must be unique, table names must be unique within a database, column names must be unique within a table, and so on.

IMHO I suggest you code table names plural. For example, “Customers”, “Orders”, “Addresses”.

- The first character must be a letter from a to z or A to Z, or the underscore (_), “at” sign (@), or number sign (#). Yes, you can use Unicode⁸ letter characters.
- Subsequent characters in an identifier can be any Unicode letter, decimal numbers, the underscore (_), “at” sign (@), or number sign (#).
- Don’t use embedded spaces in object names. Yes, you can define table and other object names that contain spaces in Access and in some of the tools (including Visual Studio), but SQL Server frowns on it. The problem is that every step along the way, you’ll wish you had taken the time to remove the spaces. Each time you define a SQL query, you’ll have to remember to bracket the long name so the TSQL parser and the development tools and wizards won’t get confused.
- Stay away from reserved words. This means you can’t use the word “Name” to refer to a customer’s name—not without taking special care⁹ when you write your TSQL queries. Every time SQL Server ships, the reserved words list grows longer. Microsoft actually includes a list of words they plan to reserve in future versions, so it’s not a good idea to use these, either. Look up “Reserved Keywords in TSQL ” in BOL to get a complete list. It’s usually (but not always) safe to concatenate two words together with an underscore, such as “Name_Like”.

⁸ Unicode Standard 2.0.

⁹ Look up “Quoted Identifiers” for more information. It’s actually easier to avoid using these names in the first place.

- Capitalization does not matter in TSQL identifiers—unless you configure your server to be case-sensitive¹⁰. I often define identifiers using *CamelCase* notation, where each word in the identifier is capitalized. This improves readability and helps get around the reserved word restrictions. Capitalization *does* matter with CLR object names—but I’ll get to that later.
- Most types of identifiers have length restrictions. If you stay under 117 characters for identifiers, you’ll stay on safe ground.

When naming stored procedures, don’t begin the name with “sp_”. Doing so tells the server that the procedure is a “system” procedure, so it takes longer to locate the object.

Tables contain one or more columns whose properties define what’s to be stored therein and how the table is to be addressed when you want to return data from the table. The basic properties include:

- **The column name (identifier):** I suggest choosing a name (without spaces) that clearly describes the contents of the column. The name cannot be longer than 128 characters. There are many schools of thought that dictate how tables and columns should be named, and if you work in a shop of any size at all, you’ll find that your development team has already settled on a standard of some kind. It could be a “Hungarian” convention that dictates that the first few letters specify the datatype (*intDaysInProduction*), or some convention that your development team has adopted. Timestamp columns should be named “*timestamp*”.
- **The column datatype:** This determines how much space the column consumes in the database, and the “type” of data it’s permitted to store. No, choosing the right datatype is not nearly as important as it was for typical DBMS implementations, as hard disks are far larger than ever; and for small databases, space considerations should be well down on the list of concerns. However, for larger databases or those with high-volume demands, reducing the size of column footprint can help performance. In any

¹⁰ It’s not necessary to configure your server in case-sensitive mode anymore. You can write individual queries or define specific columns to be case-sensitive.

case, I usually define the datatype to handle strings, numbers, graphics, or XML. I'll discuss the rudiments of choosing the right datatype in the next few pages.

- **Specifying a user-defined datatype:** When I create tables, I sometimes do so by specifying custom, user-defined datatypes. This way, I can define rules and defaults on these columns that apply to the entire database. See the discussion on UDTs, rules, and defaults later in this chapter. They are also discussed in Chapter 2.
- **If the column is permitted to accept NULL values:** If you expect to store information that might not be known as the row is added (like “Date_Married”), you need to define the column as permitting NULLs. However, you can't define a column that's going to be the table's primary key as permitting a NULL value.
- **Is the column the primary key?** If this column is the primary key (or one of the columns that together constitute the primary key), you can so indicate. You can also indicate if the PK is to be kept unique within the table.
- **Is the column value to be generated as an “Identity” value?** As described earlier in this chapter, SQL Server can automatically generate a primary key value for your table—just request that the column be designated as IDENTITY.
- **Is the column value to contain a GUID?** In this case, request that the server designate the column as ROWGUIDCOL (using the uniqueidentifier datatype) —you might want to generate the GUID yourself as new rows are added, but it's easier to use the NEWID function as the default column value.
- **How should the column be collated?** You can specify the dictionary order, case-sensitivity, and accent-sensitivity (especially if it is different from the collation specified for the database). This means you can define columns holding a name as case-sensitive and others as non-case-sensitive (or leave them to default to the database collation sequence). The collation also determines how the data is sorted. This is important for anyone working with Unicode data or character sets that don't sort the same way as the database default. See “Using SQL Collations” in BOL for more information.

- **What action should be taken when a row is deleted or updated?** By setting the ON DELETE and ON UPDATE attributes, you can get SQL Server to implement cascading deletes or updates.

Sure, there are many other options you can specify as you define your table, but the options shown here are enough to get you started. This process needs to be repeated for each column in the table and for each table in the database.

Frankly, I expect that most of you will use the Visual Studio or Management Studio tools to define tables. You'll find it's pretty easy to define your tables, primary keys, and relationships using the interactive Database Diagram tool in Visual Studio. Your other alternative is to figure out which TSQL or SMO commands are required to configure a new table (or alter an existing table). If you're getting paid by the hour, this is your best bet. All kidding aside, some folks really like the approach of creating scripts to record how their tables are defined. Fortunately, the tools can do that, too—they can take an existing database and write a file that includes all of the TSQL needed to build it up from scratch. Sure, you're going to have to add data on your own.

Using User-Defined Types, Rules, and Defaults

In SQL Server, non-CLR UDTs are pretty straightforward—they're simply aliases to the base types¹¹. This way, you can define a UDT for "PostalCode" (based on a varchar(11) and specify the PostalCode UDT when the table is created. Once defined, a UDT can be assigned a global default. That is, when a new row is added to the table and no value is supplied, SQL Server substitutes the registered default for the column value and any other columns defined with the UDT.

In a similar manner, you can also define SQL Server *rules*¹² or (better yet) *check constraints* for specific columns or to UDTs, as I discussed in Chapter 2. These constraints are used to implement your business rules—they define what's permissible in a specific column and what's

¹¹ CLR user-defined types are far more complex. I'll defer the discussion of those to Chapter 13.

¹² According to BOL, CREATE RULE will be removed in a future version of SQL Server. Microsoft suggests that I avoid using CREATE RULE in new development work and plan to modify applications that currently use it. I don't think Microsoft can drop rules anytime soon without causing a riot, so I wouldn't worry too much just yet.

not. For example, you know (based on how you run your business) that customer discounts can range from 0% to 15% and correct shipping delays are between 1 and 90 days. Setting up SQL Server rules to enforce these business rules is fairly simple—check constraints are a bit harder. Both rules and constraints can be any expression valid in a WHERE clause and can include such elements as arithmetic operators, relational operators, and predicates (for example, IN, LIKE, BETWEEN). However, the constraints cannot reference columns or other database objects. Let’s walk through the process of creating a new UDT (alias) and associated check constraints.

Start by creating a new User-Defined data type in the database by using the SQL Server Management Studio wizard that starts when you right-click on *User-defined Data Types* | *New User-defined Data Type*, as shown in Figure 3.5.

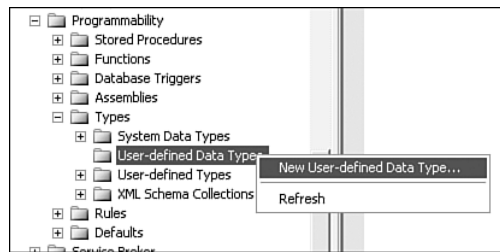


Figure 3.5 Creating a new User-Defined data type.

All I have to do is fill in the form, as shown in Figure 3.6. Here, you provide the UDT name, base datatype, and length. You can also specify that the UDT can be set to NULL. Later, I’ll use this same dialog to set the default value and the rule/check constraint for this type.

Next, I create a new constraint for our PostalCode UDT, as shown in Figure 3.7. Again, right-click the Constraints item under the selected table.

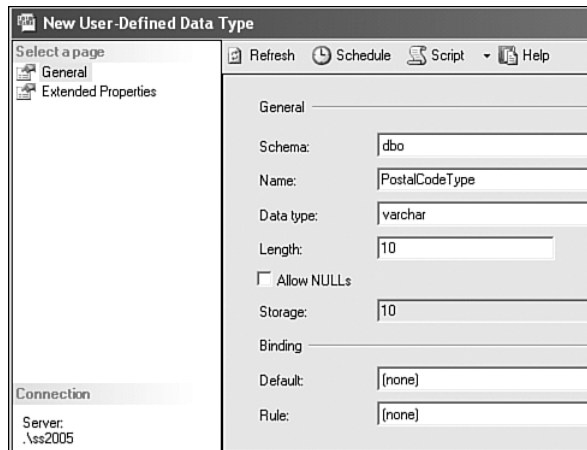


Figure 3.6 Creating a new UDT based on the varchar datatype.

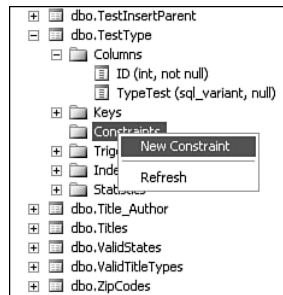


Figure 3.7 Adding a New Constraint for the PostalCode UDT.

Creating Table Indexes

Once you define your database tables and the primary key, you're going to want to add indexes to improve query performance. Without indexes, you'll find that query performance is rather slow. If you take a look at the query plan being generated, you might find that SQL Server is not fetching rows efficiently by scanning the entire table each time. Again, the interactive Database Designer tool can help set up indexes. No, don't add too many, as each index must be updated as you insert new rows. Start with an index on the primary key columns—the tools should do that for you automatically. Once you populate your database with data,

you can run the query analyzer to evaluate your indexes. This tool will tell you which indexes are helping and which are not, as well as where additional indexes will further improve performance.

Choosing the Right Data Type

When designing databases in the 1960s and 1970s, I was taught to be especially careful of how much space each data element consumed. Since hard disks were tiny by today's standards (the IBM 360 125 came with 7.25Mb to 100Mb drives),¹³ I was hard-pressed to minimize the amount of data stored in each "record." I economized by "coding" whenever and wherever I could. For example, a single column (byte) might contain several different types of data, depending on the value to be stored. When SQL Server and other relational databases were introduced, disk space was still expensive, but not nearly as much as in the mainframe days. However, more experienced database architects still choose column widths based on past experience and with the knowledge that more data means poorer performance.

Unicode vs. ANSI

In situations where you need to store data in an "international" character set, whose characters are not supported by the ANSI set, you'll have to define your columns (and string literals) as Unicode. If you take this option, SQL Server stores 16 bits for each character instead of 8. It means the same four-character entry requires 4 bytes in ANSI and 16 bytes in Unicode columns. Just remember to prefix your string literals with "N", as in N'Fred', when building Unicode expressions—Visual Studio tools and wizards do this for you if they generate the query. There is another aspect to Unicode that might surprise you. When you define a column as nvarchar, you specify a maximum length, as shown in Figure 3.8.

This DDL code allocates 50 bytes of space in the data row to the Author's name. However, this also means that the name must be no longer than 25 (16-bit) characters.

¹³ See www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP3125.html

```
CREATE TABLE [dbo].[Authors] (  
    [Au_ID] [int] IDENTITY(1,1) NOT NULL,  
    [Author] [nvarchar] (50) COLLATE SQL_Latin1_Gene  
    [Year_Born] [smallint] NULL,  
    CONSTRAINT [PK_Authors] PRIMARY KEY CLUSTERED  
(
```

Figure 3.8 Creating a table with a Unicode column.

Char vs. VarChar

I've heard the debates over use of fixed-length datatypes (like char and nchar) over variable-length types (like varchar and nvarchar). In my practices, I rarely use the fixed-width types because they're problematic in a number of respects. These types are fine for columns whose data is always the same number of characters, but if you slip and provide a value that's shorter (or longer) than the defined size, SQL Server either pads the remaining space or truncates the data (often without notice). You'll also find that it's tough to create expressions against fixed-length columns unless you match the length of both operands. For example, if your fixed-length column can contain four characters, you'll have to write an expression that has exactly four characters, or an equality expression will always return False.

```
IF MyFixedCol = 'Fred'      This returns TRUE if MyFixedCol  
                           contains "Fred".  
  
IF MyFixedCol = 'Fred '
```

But this returns FALSE.

For this reason (and others), I prefer to use variable-length types. They don't consume much extra space (if any) and when the data length varies, this approach can actually *save* space. When defining variable-length character columns, you specify the maximum amount of space to reserve for the column. This does not preallocate this space—it simply sets an upper limit. With SQL Server 2005, you can now define a varchar(max) or nvarchar(max) column that (like the TEXT datatype) can store up to 2^{31} bytes and Unicode 2^{30} bytes.

Decimal vs. Floating Point

When you record a money value in the database, it's best to understand the nature of the values you intend to store—especially the precision.

For those of you that took computer science in school, you know that it's not possible to store some values in binary. For example, $[1/3]$ is stored as .3333 (with a never-ending list of "3"s.) While the value might be close, if you add $[1/3] + [1/3] + [1/3]$, you'll get .9999—you've lost some precision. Sure, with rounding, the result is returned as 1, but in some cases, you aren't permitted to round.

IMHO In the early days of computing, clever programmers were able to strip off the extra precision (values less than a penny) and salt it away in another account. By the end of the week, they had accumulated a tidy sum—especially when millions of dollars were changing hands.

When you store a money value, be sure everyone knows the currency on which this value is based. This can help you from making a mistake when bidding on a project in the U.K., where the dollar is worth (at today's rate) about £0.529269. You might consider using a CLR-based user-defined type to keep the currency type stored with the value—especially if a single column can hold values from more than one currency.

The decimal datatype is listed (as shown in Table 1.1) under “Exact Numerics”. That is, it's designed to hold an exact value. When you declare a decimal or numeric (they are equivalent), you also can declare the *precision* and *scale* (it defaults to 18). The precision is the maximum total number of decimal digits that can be stored—including the values on either side of the decimal point. To store a value of 1234.1234, you would need a precision of 8.

The scale indicates the maximum number of decimal digits that can be stored to the right of the decimal point—this must be a value from 0 to the defined precision. The default scale is 0, so unless you define a scale, your value will be stored as a whole number (without a decimal portion). You won't be able to define a scale unless you define a precision as well. For example (as shown in Figure 3.9), to define a column with a precision of 10 and four decimal places, you would code:

```
CREATE TABLE [dbo].[Items] (  
    [ItemID] [int] IDENTITY(1,1) NOT NULL,  
    [OrderID] [int] NOT NULL,  
    [CustID] [int] NOT NULL,  
    [ISBN] [varchar] (20) COLLATE SQL_Latin1_General_CP1_CI_  
    [QtyOrdered] [int] NOT NULL CONSTRAINT [DF_Items_QtyOrd  
    [QtyShipped] [int] NOT NULL CONSTRAINT [DF_Items_QtyShi  
    [QtyBackOrdered] [int] NOT NULL,  
    [DateShipped] [datetime] NULL,  
    [Discount] [float] NOT NULL,  
    [Price] [decimal] (10, 4) NOT NULL,  
    [Notes] [nvarchar] (512) COLLATE SQL_Latin1_General_CP1_  
    [TimeStamp] [timestamp] NOT NULL,  
    CONSTRAINT [PK_Items] PRIMARY KEY CLUSTERED  
)
```

Figure 3.9 Declaring a decimal column with specific precision and scale.

Working with Imprecise Numbers

When working with scientific data where you need more precision but not 100% accuracy (which sounds a bit strange), you can choose the approximate number data types. Sure, some numbers can be expressed exactly, but others can't due to binary round-off. In the case of the *float* datatype, you can define the precision and storage size by providing a value that determines the number of bits used to store the mantissa¹⁴ of the floating point number (in scientific notation). If you supply a value between 1 and 24, the float's precision is set to 7, and it takes 4 bytes to store the value. If you provide a value between 25 and 53, the float's precision is set to 15, and it takes 8 bytes to store the value. The default is 53. Note that SQL Server 2005 resets the mantissa setting to either 1 or 53, based on the value you supply.

¹⁴ Mantissa: the fractional part of a floating-point number.

Table 3.1 SQL Server Datatypes and Their Precision

	Datatype	Bytes	
Exact Numerics			These values are stored so the value stored is expressed exactly—they are not subject to binary round-off.
Integers	bigint	8	Integer (whole number) data from -2^{63} (-9223372036854775808) through $2^{63}-1$ (9223372036854775807).
	int	4	Integer r(whole number) data from -2^{31} ($-2,147,483,648$) through $2^{31}-1$ ($2,147,483,647$).
	smallint	2	Integer data from 2^{15} ($-32,768$) through $2^{15}-1$ ($32,767$).
	tinyint	1	Integer data from 0 through 255.
Bit	bit	1	Integer data with either a 1 (True), 0 (False), or NULL value.
Decimal	decimal	5–17	Fixed precision and scale numeric data from $-10^{38}+1$ through $10^{38}-1$.
	numeric		Functionally equivalent to decimal.
Money	money	4	Monetary data values from -2^{63} ($-922,337,203,685,477.5808$) through $2^{63}-1$ ($+922,337,203,685,477.5807$), with accuracy to a ten-thousandth of a monetary unit.
	smallmoney	8	Monetary data values from $-214,748.3648$ through $+214,748.3647$, with accuracy to a ten-thousandth of a monetary unit.
Approximate Numerics			These values are stored in binary and are used when a precise but not 100% accurate value must be stored.
	float	4–8	Floating precision number data from $-1.79E+308$ through $1.79E+308$.
	doubleprecision	8	Equivalent to float(53) (8 bytes).

continues

Table 3.1 SQL Server Datatypes and Their Precision

	Datatype	Bytes	
	real	4	Floating precision number data from $-3.40E + 38$ through $3.40E + 38$.
Dates	datetime	8	Date and time data from January 1, 1753, through December 31, 9999, with an accuracy of three-hundredths of a second, or 3.33 milliseconds.
	smalldatetime	4	Date and time data from January 1, 1900, through June 6, 2079, with an accuracy of 1 minute.
ANSI Character Strings			These values are stored as strings of characters in non-Unicode (ANSI) encoding (8-bits/character).
	char	N	Fixed-length non-Unicode character data with a maximum length of 8,000 characters.
	varchar	N	Variable-length non-Unicode data with a maximum of 8,000 characters.
	varchar(max)	N	Variable-length non-Unicode data with a maximum length of $2^{31} - 1$ (2,147,483,647) characters.
	text	N	Variable-length non-Unicode data with a maximum length of $2^{31} - 1$ (2,147,483,647) characters.
Unicode Character Strings			These values are stored in Unicode (16-bits/character).
	nchar	N	Fixed-length Unicode data with a maximum length of 4,000 characters; 16 bits stored for each character.
	nvarchar	N	Variable-length Unicode data with a maximum length of 4,000 characters.

Datatype		Bytes
sysname	System-supplied user-defined data type that is functionally equivalent to nvarchar (128) and is used to reference database object names.	128
nvarchar(max)	Variable-length Unicode data with a maximum length of 2 ³⁰ – 1 (1,073,741,823) characters.	
ntext	Variable-length Unicode data with a maximum length of 2 ³⁰ – 1 (1,073,741,823) characters.	N
Binary Strings		
binary	Fixed-length binary data with a maximum length of 8,000 bytes.	N
varbinary	Variable-length binary data with a maximum length of 8,000 bytes.	N
varbinary(max)	Variable-length binary data with a maximum length of 2 ³¹ – 1 (2,147,483,647) bytes.	
image	Variable-length binary data with a maximum length of 2 ³¹ – 1 (2,147,483,647) bytes.	N
Other Types		
cursor	A reference to a server-side CURSOR.	—
sql_variant	A data type that stores values of various SQL Server-supported data types, except text, ntext, timestamp, and sql_variant.	N
table	A special data type used to store a rowset for later processing.	—
timestamp	A database-wide unique number that gets updated every time a row gets updated.	8
uniqueidentifier	A globally unique identifier (GUID).	16
xml	Names an XML schema collection. Can store up to 2GB of data.	N

Using the xml Datatype

For the first time, SQL Server 2005 introduces the new *xml* datatype. This means you're going to be able to store XML data in your table's column(s). Because *xml* is a "real" built-in type, you'll be able to use it when creating a table as a variable type, a parameter type, or a function return type. You'll also be able to use it in `CAST` or `CONVERT`. That said, I need to discuss where it makes sense to use *xml* typed data columns or *xml* typed arguments. One interesting use would permit you to pass lists of values to be used in an `IN` expression. Yes, you would need to write a function to convert this to a table-type variable.

Using the sql_variant Datatype

One of SQL Server 2000's innovations was "lifted" from Visual Basic—the "variant." The `sql_variant` datatype is unusual, in that it's designed to "morph" itself to most (non-BLOB) types. This means when you define a column as `sql_variant`, it can contain an integer (of any size), a string, a float, money, or even an *xml* structure. The `sql_variant` column value does not take on a type until you assign a value to it. I suggest you check out BOL for the rules and regulations involving this unique type.

Summary

This chapter gives you a kick-start on designing relational databases that can perform better, be easier to maintain, and be more successful. That's usually an unspoken goal of any database application project. Unless it's successful, you'll either be back working on it when you should be home relaxing or be out looking for another job—without a good recommendation from your last employer. I talked about both formal rules and informal suggestions to normalize your database. Understand that few of these rules are cast in stone, but until you fully understand them, the databases you (and your team) design and implement, populate and test, and polish and deploy won't keep bread on the table.