THE EXPERT'S VOICE® IN SQL SERVER

Pro SQL Server 2012 Relational Database Design and Implementation

COMBINE RELATIONAL DATABASE BEST PRACTICES WITH THE LATEST IN SQL SERVER IMPLEMENTATION FEATURES

Louis Davidson with Jessica M. Moss

Apress®

www.it-ebooks.info

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Apress®

www.it-ebooks.info

Contents at a Glance

Foreword xix

About the Author xxi

About the Technical Reviewer xxiii

Acknowledgments xxv

Introduction xxvii

■ Chapter 1: The Fundamentals 1

■ Chapter 2: Introduction to Requirements 37

■ Chapter 3: The Language of Data Modeling 53

■ Chapter 4: Initial Data Model Production 91

■ Chapter 5: Normalization 129

■ Chapter 6: Physical Model Implementation Case Study 169

■ Chapter 7: Data Protection with Check Constraints and Triggers 245

■ Chapter 8: Patterns and Anti-Patterns 301

■ Chapter 9: Database Security and Security Patterns 371

■ Chapter 10: Table Structures and Indexing 445

■ Chapter 11: Coding for Concurrency 505

■ Chapter 12: Reusable Standard Database Components 563

www.it-ebooks.info

■ Chapter 13: Considering Data Access Strategies 595
■ Chapter 14: Reporting Design 639
■ Appendix A 671
■ Appendix B 707

Index 735

www.it-ebooks.info

Introduction

I often ask myself, "Why do I do this? Why am I writing another edition of this book? Is it worth it? Isn't there anything else that I could be doing that would be more beneficial to me, my family, or the human race? Well, of course there is. The fact is, however, I generally love relational databases, I love to write, and I want to help other people get better at what they do.

When I was first getting started designing databases, I learned from a few great mentors, but as I wanted to progress, I started looking for material on database design, and there wasn't much around. The best book I found was an edition of Chris Date's *An Introduction to Database Systems* (Addison Wesley, 2003), and I read as much as I could comprehend. The problem, however, was that I quickly got lost and started getting frustrated that I couldn't readily translate the theory of it all into a design process that really seems quite simple once you get down to it. I really didn't get it until I had spent years designing databases, failing over and over until I finally saw the simplicity of it all. In Chris's book, as well as other textbooks I had used, it was clear that a lot of theory, and even more math, went into creating the relational model.

If you want a deep understanding or relational theory, Chris's book is essential reading, along with lots of other books (Database Debunkings, www.dbdebunk.com/books.html, is a good place to start looking for more titles). The problem is that most of these books have far more theory than the average practitioner wants (or will take the time to read), and they don't really get into the actual implementation on an actual database system. My book's goal is simply to fill that void and bridge the gap between academic textbooks and the purely implementation-oriented books that are commonly written on SQL Server. My intention is not to knock those books, not at all—I have numerous versions of those types of books on my shelf. This book is more of a technique-oriented book than a how-to book teaching you the features of SQL Server. I will cover many the most typical features of the relational engine, giving you techniques to work with. I can't, however, promise that this will be the only book you need on your shelf.

If you have previous editions of this book, you might question why you need this next edition, and I ask myself that every time I sit down to work on the next edition. You might guess that the best reason is that I cover the new SQL Server 2012 features. Clearly that is a part of it, but the base features in the relational engine that you need to know to design and implement most databases is not changing tremendously over time. Under the covers, the engine has taken more leaps, and hardware has continued up and up as the years progress. The biggest changes to SQL Server 2012 for the relational programmer lie in some of the T-SQL language features, like windowing functions that come heavily into play for the programmer that will interact with your freshly designed and loaded databases.

No, the best reason to buy the latest version of the book is that I continue to work hard to come up with new content to make your job easier. I've reworked the chapter on normalization to be easier to understand, added quite a few more patterns of development to Chapter 7, included a walkthrough of the development process (including testing) in Chapter 6, some discussion about the different add-ins you can use to enhance your databases, and generally attempted to improve the entire book throughout to be more concise (without losing the folksy charm, naturally). Finally, I added a chapter about data warehousing, written by a great friend and fellow MVP Jessica Moss.

www.it-ebooks.info

Oscar Wilde, the poet and playwright, once said, "I am not young enough to know everything." It is with some chagrin that I must look back at the past and realize that I thought I knew everything just before I wrote my first book, *Professional SQL Server 2000 Database Design* (Wrox Press, 2001). It was ignorant, unbridled, unbounded enthusiasm that gave me the guts to write the first book. In the end, I did write that first edition, and it was a decent enough book, largely due to the beating I took from my technical editing staff. And if I hadn't possessed such enthusiasm initially, I would not be likely to be writing this fifth edition of the book. However, if you had a few weeks to burn and you went back and compared each edition of this book, chapter by chapter, section by section, to the current edition, you would notice a progression of material and a definite maturing of the writer.

There are a few reasons for this progression and maturity. One reason is the editorial staff I have had over the past three versions: first Tony Davis and now Jonathan Gennick. Both of them were very tough on my writing style and did wonders on the structure of the book. Another reason is simply experience, as over eight years have passed since I started the first edition. But most of the reason that the material has progressed is that it's been put to the test. While I have had my share of nice comments, I have gotten plenty of feedback on how to improve things (some of those were not-nice comments!). And I listened very intently, keeping a set of notes that start on the release date. I am always happy to get any feedback that I can use (particularly if it doesn't involve any anatomical terms for where the book might fit). I will continue to keep my e-mail address available (louis@drsql.org), and you can leave anonymous feedback on my web site if you want (drsql.org). You may also find an addendum there that covers any material that I may uncover that I wish I had known at the time of this writing.

Purpose of Database Design

What is the purpose of database design? Why the heck should you care? The main reason is that a properly designed database is straightforward to work with, because everything is in its logical place, much like a well-organized cupboard. When you need paprika, it's easier to go to the paprika slot in the spice rack than it is to have to look for it everywhere until you find it, but many systems are organized just this way. Even if every item has an assigned place, of what value is that item if it's too hard to find? Imagine if a phone book wasn't sorted at all. What if the dictionary was organized by placing a word where it would fit in the text? With proper organization, it will be almost instinctive where to go to get the data you need, even if you have to write a join or two. I mean, isn't that fun after all?

You might also be surprised to find out that database design is quite a straightforward task and not as difficult as it may sound. Doing it right is going to take more up-front time at the beginning of a project than just slapping a database as you go along, but it pays off throughout the full life cycle of a project. Of course, because there's nothing visual to excite the client, database design is one of the phases of a project that often gets squeezed to make things seem to go faster. Even the least challenging or uninteresting user interface is still miles more interesting to the average customer than the most beautiful data model. Programming the user interface takes center stage, even though the data is generally why a system gets funded and finally created. It's not that your colleagues won't notice the difference between a cruddy data model and one that's a thing of beauty. They certainly will, but the amount of time required to decide the right way to store data correctly can be overlooked when programmers need to code. I wish I had an answer for that problem, because I could sell a million books with just that. This book will assist you with some techniques and processes that will help you through the process of designing databases, in a way that's clear enough for novices and helpful to even the most seasoned professional.

This process of designing and architecting the storage of data belongs to a different role to those of database setup and administration. For example, in the role of data architect, I seldom create users, perform backups, or set up replication or clustering. Little is mentioned of these tasks, which are considered administration and the role of the DBA. It isn't uncommon to wear both a developer hat and a DBA hat (in fact, when you work in a smaller organization, you may find that you wear so many hats your neck tends to hurt), but your designs will generally be far better thought out if you can divorce your mind from the more implementation-bound roles that make you wonder how hard it will be to use the data. For the most part, database design looks harder than it is.

www.it-ebooks.info

Who This Book Is For

This book is written for professional programmers who have the need to design a relational database using any of the Microsoft SQL Server family of databases. It is intended to be useful for the beginner to advanced programmer, either strictly database programmers or a programmer that has never used a relational database product before to learn why relational databases are designed in the way they are, and get some practical examples and advice for creating databases. Topics covered cater to the uninitiated to the experienced architect to learn techniques for concurrency, data protection, performance tuning, dimensional design, and more.

How This Book Is Structured

This book is comprised of the following chapters, with the first five chapters being an introduction to the fundamental topics and process that one needs to go through/know before designing a database. Chapters 6 is an exercise in learning how a database is put together using scripts, and the rest of the book is taking topics of design and implementation and providing instruction and lots of examples to help you get started building databases.

Chapter 1: The Fundamentals. This chapter provides a basic overview of essential terms and concepts necessary to get started with the process of designing a great relational database.

Chapter 2: Introduction to Requirements. This chapter provides an introduction to how to gather and interpret requirements from a client. Even if it isn't your job to do this task directly from a client, you will need to extract some manner or requirements for the database you will be building from the documentation that an analyst will provide to you.

Chapter 3: The Language of Data Modeling. This chapter serves as the introduction to the main tool of the data architect—the model. In this chapter, I introduce one modeling language (IDEF1X) in detail, as it's the modeling language that's used throughout this book to present database designs. I also introduce a few other common modeling languages for those of you who need to use these types of models for preference or corporate requirements.

Chapter 4: Initial Data Model Production. In the early part of creating a data model, the goal is to discuss the process of taking a customer's set of requirements and to put the tables, columns, relationships, and business rules into a data model format where possible. Implementability is less of a goal than is to faithfully represent the desires of the eventual users.

Chapter 5: Normalization. The goal of normalization is to make your usage of the data structures that get designed in a manner that maps to the relational model that the SQL Server engine was created for. To do this, we will take the set of tables, columns, relationships, and business rules and format them in such a way that every value is stored in one place and every table represents a single entity. Normalization can feel unnatural the first few times you do it, because instead of worrying about how you'll use the data, you must think of the data and how the structure will affect that data's quality. However, once you mastered normalization, not to store data in a normalized manner will feel wrong.

Chapter 6: Physical Model Implementation Case Study. In this chapter, we will walk through the entire process of taking a normalized model and translating it into a working database. This is the first point in the database design process in which we fire up SQL Server and start building scripts to build database objects. In this chapter, I cover building tables—including choosing the datatype for columns—as well as relationships.

Chapter 7: Data Protection with CHECK Constraints and Triggers. Beyond the way data is arranged in tables and columns, other business rules may need to be enforced. The front line of defense for enforcing data integrity conditions in SQL Server is formed by CHECK constraints and triggers, as users cannot innocently avoid them.

www.it-ebooks.info

Chapter 8: Patterns and Anti-Patterns. Beyond the basic set of techniques for table design, there are several techniques that I use to apply a common data/query interface for my future convenience in queries and usage. This chapter will cover several of the common useful patterns as well as take a look at some patterns that some people will use to make things easier to implement the interface that can be very bad for your query needs.

Chapter 9: Database Security and Security Patterns. Security is high in most every programmer's mind these days, or it should be. In this chapter, I cover the basics of SQL Server security and show how to employ strategies to use to implement data security in your system, such as employing views, triggers, encryption, and even using SQL Server Profiler.

Chapter 10: Table Structures and Indexing. In this chapter, I show the basics of how data is structured in SQL Server, as well as some strategies for indexing data for better performance.

Chapter 11: Coding for Concurrency. As part of the code that's written, some consideration needs to be taken when you have to share resources. In this chapter, I describe several strategies for how to implement concurrency in your data access and modification code.

Chapter 12: Reusable Standard Database Components. In this chapter, I discuss the different types of reusable objects that can be extremely useful to add to many (if not all) of your databases you implement to provide a standard problem solving interface for all of your systems while minimizing inter-database dependencies

Chapter 13: Considering Data Access Strategies. In this chapter, the concepts and concerns of writing code that accesses SQL Server are covered. I cover ad hoc SQL versus stored procedures (including all the perils and challenges of both, such as plan parameterization, performance, effort, optional parameters, SQL injection, and so on), as well as discuss whether T-SQL or CLR objects are best.

Chapter 14: Reporting Design. Written by Jessica Moss, this chapter presents an overview of how designing for reporting needs differs from OLTP/relational design, including an introduction to dimensional modeling used for data warehouse design.

Appendix A: Scalar Datatype Reference. In this appendix, I present all of the types that can be legitimately considered scalar types, along with why to use them, their implementation information, and other details.

Appendix B: DML Trigger Basics and Templates. Throughout the book, triggers are used in several examples, all based on a set of templates that I provide in this appendix, including example tests of how they work and tips and pointers for writing effective triggers.

Prerequisites

The book assumes that the reader has some experience with SQL Server, particularly writing queries using existing databases. Beyond that, most concepts that are covered will be explained and code should be accessible to anyone with an experience programming using any language.

Downloading the Code

A download will be available as a Management Studio project and as individual files from the Apress download site. Files will also be available from my web site, http://drsql.org/ProSQLServerDatabaseDesign.aspx, as well as links to additional material I may make available between now and any future editions of the book.

www.it-ebooks.info

Contacting the Authors

Don't hesitate to give me feedback on the book, anytime, at my web site (drsql.org) or my e-mail (louis@drsql.org). I'll try to improve any sections that people find lacking and publish them to my blog (http://sqlblog.com/blogs/louis_davidson) with the tag DesignBook, as well as to my web site (http://drsql.org/ProSQLServerDatabaseDesign.aspx). I'll be putting more information there, as it becomes available, pertaining to new ideas, goof-ups I find, or additional materials that I choose to publish because I think of them once this book is no longer a jumble of bits and bytes and is an actual instance of ink on paper.

www.it-ebooks.info

CHAPTER 1

The Fundamentals

A successful man is one who can lay a firm foundation with the bricks others have thrown at him.

—David Brinkley

Face it, education in fundamentals is rarely something that anyone considers exactly fun, at least unless you already have a love for the topic in some level. In elementary school, there were fun classes, like recess and lunch for example. But when handwriting class came around, very few kids really liked it, and most of those who did just loved the taste of the pencil lead. But handwriting class was an important part of childhood educational development. Without it, you wouldn't be able to write on a white board and without that skill could you actually stay employed as a programmer? I know I personally am addicted to the smell of whiteboard marker, which might explain more than my vocation.

Much like handwriting was an essential skill for life, database design has its own set of skills that you need to get under your belt. While database design is not a hard skill to learn, it is not exactly a completely obvious one either. In many ways, the fact that it isn't a hard skill makes it difficult to master. Databases are being designed all of the time by people of all skill levels. Administrative assistants build databases using Excel; newbie programmers do so with Access and even SQL Server over and over, and they rarely are 100% wrong. The problem is that in almost every case the design produced is fundamentally flawed, and these flaws are multiplied during the course of implementation; they may actually end up requiring the user to do far more work than necessary and cause future developers even more pain. When you are finished with this book, you should be able to design databases that reduce the effects of common fundamental blunders. If a journey of a million miles starts with a single step, the first step in the process of designing quality databases is understanding why databases are designed the way they are, and this requires us to cover the fundamentals.

I know this topic may bore you, but would you drive on a bridge designed by an engineer who did not understand physics? Or would you get on a plane designed by someone who didn't understand the fundamentals of flight? Sounds quite absurd, right? So, would you want to store your important data in a database designed by someone who didn't understand the basics of database design?

The first five chapters of this book are devoted to the fundamental tasks of relational database design and preparing your mind for the task at hand: designing databases. The topics won't be particularly difficult in nature, and I will do my best to keep the discussion at the layman's level, and not delve so deeply that you punch me if you pass me in the hall at the SQL PASS Summit [www.sqlpass.org]. For this chapter, we will start out looking at the basic background topics that are so very useful.

www.it-ebooks.info

- *History*: Where did all of this relational database stuff come from? In this section I will present some history, largely based on Codd's 12 Rules as an explanation for why the RDBMS (Relational Database Management System) is what it is.
- *Relational data structures*: This section will provide concise introductions of some of the fundamental database objects, including the database itself, as well as tables, columns, and keys. These objects are likely familiar to you, but there are some common misunderstandings in their usage that can make the difference between a mediocre design and a high-class, professional one. In particular, misunderstanding the vital role of keys in the database can lead to severe data integrity issues and to the mistaken belief that such keys and constraints can be effectively implemented outside the database. (Here is a subtle hint: they can't.)
- *Relationships between entities*: We will briefly survey the different types of relationships that can exist between relational the relational data structures introduced in the relational data structures section.
- *Dependencies*: The concept of dependencies between values and how they shape the process of designing databases later in the book will be discussed
- *Relational programming*: This section will cover the differences between functional programming using C# or VB (Visual Basic) and relational programming using SQL (Structured Query Language).
- *Database design phases*: This section provides an overview of the major phases of relational database design: conceptual/logical, physical, and storage. For time and budgetary reasons, you might be tempted to skip the first database design phase and move straight to the physical implementation phase. However, skipping any or all of these phases can lead to an incomplete or incorrect design, as well as one that does not support high-performance querying and reporting.

At a minimum, this chapter on fundamentals should get us to a place where we have a set of common terms and concepts to use throughout this book when discussing and describing relational databases. Some of these terms are misunderstood and misused by a large number (if not a majority) of people. If we are not in agreement on their meaning from the beginning, eventually you might end up wondering what the heck we're talking about. Some might say that semantics aren't worth arguing about, but honestly, they are the *only* thing worth arguing about. Agreeing to disagree is fine if two parties understand one another, but the true problems in life tend to arise when people are in complete agreement about an idea but disagree on the terms used to describe it.

Among the terms that need introduction is modeling, specifically data modeling. Modeling is the process of capturing the essence of a system in a known language that is common to the user. A data model is a specific type of model that focuses entirely on the storage and management of the data storage medium, reflecting all of the parts of a database. It is a tool that we will use throughout the process from documentation to the end of the process where users have a database. The term "modeling" is often used as a generic term for the overall process of creating a database. As you can see from this example, we need to get on the same page when it comes to the concepts and basic theories that are fundamental to proper database design.

Taking a Brief Jaunt Through History

No matter what country you hail from, there is, no doubt, a point in history when your nation began. In the United States, that beginning came with the Declaration of Independence, followed by the Constitution of the United States (and the ten amendments known as the Bill of Rights). These documents are deeply ingrained

www.it-ebooks.info

in the experience of any good citizen of the United States. Similarly, we have three documents that are largely considered the start of relational databases.

In 1979, Edgar F Codd, who worked for the IBM Research Laboratory at the time, wrote a paper entitled "A Relational Model of Data For Large Shared Data Banks," which was printed in *Communications of the ACM* ("ACM" is the Association for Computing Machinery [www.acm.org]). In this 11-page paper, Codd introduces a revolutionary idea for how to break the physical barriers of the types of databases in use at that time. Then, most database systems were very structure oriented, requiring a lot of knowledge of how the data was organized in the storage. For example, to use indexes in the database, specific choices would be made, like only indexing one key, or if multiple indexes existed, the user were required to know the name of the index to use it in a query.

As most any programmer knows, one of the fundamental tenets of good programming is to attempt low coupling of different computer subsystem, and needing to know about the internal structure of the data storage was obviously counterproductive. If you wanted to change or drop an index, the software and queries that used the database would also need to be changed. The first half of the Codd's relational model paper introduced a set of constructs that would be the basis of what we know as a relational database. Concepts such as tables, columns, keys (primary and candidate), indexes, and even an early form of normalization are included. The second half of the paper introduced set-based logic, including joins. This paper was pretty much the database declaration of storage independence.

Moving six years in the future, after companies began to implement supposed relational database systems, Codd wrote a two-part article published by *Computerworld* magazine entitled "Is Your DBMS Really Relational?" and "Does Your DBMS Run By the Rules?" on October 14 and October 21, 1985. Though it is nearly impossible to get a copy of these original articles, many web sites outline these rules, and I will too. These rules go beyond relational theory and define specific criteria that need to be met in an RDBMS, if it's to be truly be considered relational.

After introducing Codd's rules, I will touch very briefly on the different standards as they have evolved over the years.

Introducing Codd's Rules for an RDBMS

I feel it is useful to start with Codd's rules, because while these rules are now 27 years old, they do probably the best job of setting up not only the criteria that can be used to measure how relational a database is but also the reasons why relational databases are implemented as they are. The neat thing about these rules is that they are seemingly just a formalized statement of the KISS manifesto for database users—keep it simple stupid, or keep it standard, either one. By establishing a formal set of rules and principles for database vendors, users could access data that was not only simplified from earlier data platforms but worked pretty much the same on any product that claimed to be relational. Of course, things are definitely not perfect in the world, and these are not the final principles to attempt to get everyone on the same page. Every database vendor has a different version of a relational engine, and while the basics are the same, there are wild variations in how they are structured and used. The basics are the same, and for the most part the SQL language implementations are very similar (I will discuss very briefly the standards for SQL in the next section). The primary reason that these rules are so important for the person just getting started with design is that they elucidate why SQL Server and other relational engine based database systems work the way they do.

Rule 1: The Information Principle

> *All information in the relational database is represented in exactly one and only one way—by values in tables.*

www.it-ebooks.info

While this rule might seem obvious after just a little bit of experience with relational databases, it really isn't. Designers of database systems could have used global variables to hold data or file locations or come up with any sort of data structure that they wanted. Codd's first rule set the goal that users didn't have to think about where to go to get data. One data structure—the table—followed a common pattern rows and columns of data that users worked with.

Many different data structures were in use back then that required a lot of internal knowledge of data. Think about all of the different data structures and tools you have used. Data could be stored in files, a hierarchy (like the file system), or any method that someone dreamed of. Even worse, think of all of the computer programs you have used; how many of them followed a common enough standard that they work just like everyone else's? Very few, and new innovations are coming every day.

While innovation is rarely a bad thing, innovation in relational databases is ideally limited to the layer that is encapsulated from the user's view. The same database code that worked 20 years ago could easily work today with the simple difference that it now runs a great deal faster. There have been advances in the language we use (SQL), but it hasn't changed tremendously because it just plain works.

Rule 2: Guaranteed Access

> *Each and every datum (atomic value) is guaranteed to be logically accessible by resorting to a combination of table name, primary key value, and column name.*

This rule is an extension of the first rule's definition of how data is accessed. While all of the terms in this rule will be defined in greater detail later in this chapter, suffice it to say that columns are used to store individual points of data in a row of data, and a primary key is a way of uniquely identifying a row using one or more columns of data. This rule defines that, at a minimum, there will be a non-implementation-specific way to access data in the database. The user can simply ask for data based on known data that uniquely identifies the requested data. "Atomic" is a term that we will use frequently; it simply means a value that cannot be broken down any further without losing its fundamental value. It will be covered several more times in this chapter and again in Chapter 5 when we cover normalization.

Together with the first rule, rule two establishes a kind of addressing system for data as well. The table name locates the correct table; the primary key value finds the row containing an individual data item of interest, and the column is used to address an individual piece of data.

Rule 3: Systematic Treatment of NULL Values

> *NULL values (distinct from empty character string or a string of blank characters and distinct from zero or any other number) are supported in the fully relational RDBMS for representing missing information in a systematic way, independent of data type.*

Good grief, if there is one topic I would have happily avoided in this book, it is missing values and how they are implemented with NULLs. NULLs are the most loaded topic of all because they are so incredibly different to use than all other types of data values you will encounter, and they are so often interpreted and used wrong. However, if we are going to broach the subject sometime, we might as well do so now.

The NULL rule requires that the RDBMS support a method of representing "missing" data the same way for every implemented datatype. This is really important because it allows you to indicate that you have no value for every column consistently, without resorting to tricks. For example, assume you are making a list of how many computer mice you have, and you think you still have an Arc mouse, but you aren't sure. You list Arc mouse to

www.it-ebooks.info

let yourself know that you are interested in such mice, and then in the count column you put—what? Zero? Does this mean you don't have one? You could enter –1, but what the heck does that mean? Did you loan one out? You could put "Not sure" in the list, but if you tried to programmatically sum the number of mice you have, you will have to deal with "Not sure."

To solve this problem, the placeholder NULL was devised to work regardless of datatype. For example, in string data, NULLs are distinct from an empty character string, and they are always to be considered a value that is unknown. Visualizing them as UNKNOWN is often helpful to understanding how they work in math and string operations. NULLs propagate through mathematic operations as well as string operations. NULL + <anything> = NULL, the logic being that NULL means "unknown." If you add something known to something unknown, you still don't know what you have; it's still unknown. Throughout the history of relational database systems, NULLs have been implemented incorrectly or abused, so there are generally settings to allow you to ignore the properties of NULLs. However, doing so is inadvisable. NULL values will be a topic throughout this book; for example, we deal with patterns for missing data in Chapter 8, and in many other chapters, NULLs greatly affect how data is modeled, represented, coded, and implemented. Like I said, NULLs are painful but necessary.

Rule 4: Dynamic Online Catalog Based on the Relational Model

> *The database description is represented at the logical level in the same way as ordinary data, so authorized users can apply the same relational language to its interrogation as they apply to regular data.*

This rule requires that a relational database be self-describing. In other words, the database must contain tables that catalog and describe the structure of the database itself, making the discovery of the structure of the database easy for users, who should not need to learn a new language or method of accessing metadata. This trait is very common, and we will make use of the system catalog tables regularly throughout the latter half of this book to show how something we have just implemented is represented in the system and how you can tell what other similar objects have also been created.

Rule 5: Comprehensive Data Sublanguage Rule

> *A relational system may support several languages and various modes of terminal use. However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and whose ability to support all of the following is comprehensible: a. data definition b. view definition c. data manipulation (interactive and by program) d. integrity constraints e. authorization f. transaction boundaries (begin, commit, and rollback).*

This rule mandates the existence of a relational database language, such as SQL, to manipulate data. The language must be able to support all the central functions of a DBMS: creating a database, retrieving and entering data, implementing database security, and so on. SQL as such isn't specifically required, and other experimental languages are in development all of the time, but SQL is the de facto standard relational language and has been in use for over 20 years.

Relational languages are different from procedural (and most other types of) languages, in that you don't specify how things happen, or even where. In ideal terms, you simply ask a question of the relational engine, and it does the work. You should at least, by now, realize that this encapsulation and relinquishing of responsibilities is a very central tenet of relational database implementations. Keep the interface simple and encapsulated from the

www.it-ebooks.info

realities of doing the hard data access. This encapsulation is what makes programming in a relational language very elegant but oftentimes frustrating. You are commonly at the mercy of the engine programmer, and you cannot implement your own access method, like you could in C# if you discovered an API that wasn't working well. On the other hand, the engine designers are like souped up rocket scientists and, in general, do an amazing job of optimizing data access, so in the end, it is better this way, and Grasshopper, the sooner you release responsibility and learn to follow the relational ways, the better.

Rule 6: View Updating Rule

> *All views that are theoretically updateable are also updateable by the system.*

A table, as we briefly defined earlier, is a structure with rows and columns that represents data stored by the engine. A view is a stored representation of the table that, in itself, is technically a table too; it's commonly referred to as a virtual table. Views are generally allowed to be treated just like regular (sometimes referred to as materialized) tables, and you should be able to create, update, and delete data from a view just like a from table. This rule is really quite hard to implement in practice because views can be defined in any way the user wants, but the principle is a very useful nonetheless.

Rule 7: High-Level Insert, Update, and Delete

> *The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update, and deletion of data.*

This rule is probably the biggest blessing to programmers of them all. If you were a computer science student, an adventurous hobbyist, or just a programming sadist like the members of the Microsoft SQL Server Storage Engine team, you probably had to write some code to store and retrieve data from a file. You will probably also remember that it was very painful and difficult to do, and usually you were just doing it for a single user. Now, consider simultaneous access by hundreds or thousands of users to the same file and having to guarantee that every user sees and is able to modify the data consistently and concurrently. Only a truly excellent system programmer would consider that a fun challenge.

Yet, as a relational engine user, you write very simple statements using SELECT, INSERT, UPDATE, and DELETE statements that do this every day. Writing these statements is like shooting fish in a barrel—extremely easy to do (it's confirmed by *Mythbusters* as easy to do, if you are concerned, but don't shoot fish in a barrel unless you are planning on having fish for dinner—it is not a nice thing to do). Simply by writing a single statement using a known table and its columns, you can put new data into a table that is also being used by other users to view, change data, or whatever. In Chapter 11, we will cover the concepts of concurrency to see how this multitasking of modification statements is done, but even the concepts we cover there can be mastered by us common programmers who do not have a PhD from MIT.

Rule 8: Physical Data Independence

> *Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representation or access methods.*

www.it-ebooks.info

Applications must work using the same syntax, even when changes are made to the way in which the database internally implements data storage and access methods. This rule basically states that the way the data is stored must be independent of the manner in which it's used and the way data is stored is immaterial to the users. This rule will play a big part of our entire design process, because we will do our best to ignore implementation details and design for the data needs of the user.

Rule 9: Logical Data Independence

> *Application programs and terminal activities remain logically unimpaired when information preserving changes of any kind that theoretically permit unimpairment are made to the base tables.*

While rule eight was concerned with the internal data structures that interface the relational engine to the file system, this rule is more centered with things we can do to the table definition in SQL. Say you have a table that has two columns, A and B. User X makes use of A; user Y uses A and B. If the need for a column C is discovered, adding column C should not impair users X's and Y's programs at all. If the need for column B was eliminated, and hence the column was removed, it is acceptable that user Y would then be affected, yet user X, who only needed column A, would still be unaffected.

As a quick aside, there is a construct known as star (*) that is used as a wildcard for all of the columns in the table (as in SELECT * FROM table). The principals of logical data independence are largely the reason why we avoid getting all of the columns like this for anything other than nonreusable ad hoc access (like a quick check to see what data is in the table to support a user issue). Using this construct tightly couples the entire table to the user, whether or not a new column is added. This new column may in fact be unneeded (and contain a huge amount of data!) or a unnecessary column might be removed but then break your code unexpectedly. Declaring exactly the data you need and expect is a very good plan in your code that you write for reuse.

Rule 10: Integrity Independence

> *Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.*

Another of the truly fundamental concepts stressed by the founder of the relational database concepts was that data should have integrity; in other words, it's important for the system to protect itself from data issues. Predicates that state that data must fit into certain molds were to be implemented in the database. Minimally, the RDBMS must internally support the definition and enforcement of entity integrity (primary keys) and referential integrity (foreign keys). We also have unique constraints to enforce keys that aren't the primary key, NULL constraints to state whether or not a value must be known when the row is created, as well as check constraints that are simply table or column conditions that must be met. For example, say you have a column that stores employees' salaries. It would be good to add a condition to the salary storage location to make sure that the value is greater than or equal to zero, because you may have unpaid volunteers, but I can only think of very few jobs where you pay to work at your job.

This rule is just about as controversial at times as the concept of NULLs. Application programmers don't like to give up control of the management of rules because managing the general rules in a project becomes harder. On the other hand, many types of constraints you need to use the engine for are infeasible to implement in the application layer (uniqueness and foreign keys are two very specific examples, but any rule that reaches outside of the one row of data cannot be done both quickly and safely in the application layer because of the rigors of concurrent user access).

www.it-ebooks.info

The big takeaway for this particular item should be that the engine provides tools to protect data, and in the least intrusive manner possible, you should use the engine to protect the integrity of the data.

Rule 11: Distribution Independence

> *The data manipulation sublanguage of a relational DBMS must enable application programs and terminal activities to remain logically unimpaired whether and whenever data are physically centralized or distributed.*

This rule was exceptionally forward thinking in 1985 and is still only getting close to being realized for anything but the largest systems. It is very much an extension of the physical independence rule taken to a level that spans the containership of a single computer system. If the data is moved to a different server, the relational engine should recognize this and just keep working.

Rule 12: Nonsubversion Rule

> *If a relational system has or supports a low-level (single-record-at-a-time) language, that low-level language cannot be used to subvert or bypass the integrity rules or constraints expressed in the higher-level (multiple-records-at-a-time) relational language.*

This rule requires that alternate methods of accessing the data are not able to bypass integrity constraints, which means that users can't violate the rules of the database in any way. Generally speaking, at the time of this writing, most tools that are not SQL based do things like check the consistency of the data and clean up internal storage structures. There are also row-at-a-time operators called cursors that deal with data in a very nonrelational manner, but in all cases, they do not have the capability to go behind or bypass the rules of the RDBMS.

A common big cheat is to bypass rule checking when loading large quantities of data using bulk loading techniques. All of the integrity constraints you put on a table generally will be quite fast and only harm performance an acceptable amount during normal operations. But when you have to load millions of rows, doing millions of checks can be very expensive, and hence there are tools to skip integrity checks. Using a bulk loading tool is a necessary evil, but it should never be an excuse to allow data with poor integrity into the system.

Nodding at SQL Standards

In addition to Codd's rules, one topic that ought to be touched on briefly is the SQL standards. Rules five, six, and seven all pertain to the need for a high level language that worked on data in a manner encapsulated the nasty technical details from the user. Hence, the SQL language was born. The language SQL was initially called SEQUEL (**S**tructured **E**nglish **Que**ry **L**anguage), but the name was changed to SQL for copyright reasons. However, it is still often pronounced "sequel" today (sometimes, each letter is pronounced separately). SQL had its beginnings in the early 1970s with Donald Chamberlin and Raymond Boyce (see http://en.wikipedia.org/wiki/SQL), but the path to get us to the place where we are now was quite a trip. Multiple SQL versions were spawned, and the idea of making SQL a universal language was becoming impossible.

In 1986, the American National Standards Institute (ANSI), created a standard called SQL-86 for how the SQL language should be moved forward. This standard took features that the major players at the time had been implementing in an attempt to make code interoperable between these systems, with the engines being

www.it-ebooks.info

the part of the system that would be specialized. This early specification was tremendously limited and did not even include referential integrity constraints. In 1989, the SQL-89 specification was adopted, and it included referential integrity, which was a tremendous improvement and a move toward implementing Codd's twelfth rule (see *Handbook on Architectures of Information Systems* by Bernus, Mertins, and Schmidt [Springer 2006]).

Several more versions of the SQL standard have come and gone, in 1992, 1999, 2003, 2006, and 2008. For the most part, these documents are not exactly easy reading, nor do they truly mean much to the basic programmer/practitioner, but they can be quite interesting in terms of putting new syntax and features of the various database engines into perspective. The standard also helps you to understand what people are talking about when they talk about standard SQL. The standard also can help to explain some of the more interesting choices that are made by database vendors.

This brief history lesson was mostly for getting you started to understand why relational database are implemented as they are today. In three papers, Codd took a major step forward in defining what a relational database is and how it is supposed to be used. In the early days, Codd's 12 rules were used to determine whether a database vendor could call itself relational and presented stiff implementation challenges for database developers. As you will see by the end of this book, even today, the implementation of the most complex of these rules is becoming achievable, though SQL Server (and other RDBMSs) still fall short of achieving their objectives. Plus, the history of the SQL language has been a very interesting one as standards committees from various companies come together and try to standardize the stuff they put into their implementations (so everyone else gets stuck needing to change).

Obviously, there is a lot more history between 1985 and today. Many academics including Codd himself, C. J. Date, and Fabian Pascal (both of whom contribute to their site http://www.dbdebunk.com), Donald Chamberlin, Raymond Boyce (who contributed to one of the Normal Forms, covered in Chapter 6), and many others have advanced the science of relational databases to the level we have now. Some of their material is interesting only to academics, but most of it has practical applications even if it can be very hard to understand, and it's very useful to anyone designing even a modestly complex model. I definitely suggest reading as much of their material, and all the other database design materials, as you can get your hands on after reading this book (after, read: after). In this book, we will keep everything at a very practical level that is formulated to cater to the general practitioner to get down to the details that are most important and provide common useful constructs to help you start developing great databases quickly.

Recognizing Relational Data Structures

This section introduces the following core relational database structures and concepts:

- Database and schema
- Tables, rows, and columns
- Missing values (nulls)
- Uniqueness constraints (keys)

As a person reading this book, this is probably not your first time working with a database, and therefore, you are no doubt somewhat familiar with some of these concepts. However, you may find there are at least a few points presented here that you haven't thought about that might help you understand why we do things later—for example, the fact that a table consists of unique rows or that within a single row a column must represent only a single value. These points make the difference between having a database of data that the client relies on without hesitation and having one in which the data is constantly challenged.

Note, too, that in this section we will only be talking about items from the relational model. In SQL Server, you have a few layers of containership based on how SQL Server is implemented. For example, the concept of a server is analogous to a computer, or a virtual machine perhaps. On a server, you may have multiple instances of SQL Server that can then have multiple databases. The terms "server" and "instance" are often misused as

www.it-ebooks.info

synonyms, mostly due to the original way SQL Server worked allowing only a single instance per server (and since the name of the product *is* SQL Server, it is a natural mistake). For most of this book, we will not need to look at any higher level than the database, which I will introduce in the following section.

Introducing Databases and Schemas

A database is simply a structured collection of facts or data. It needn't be in electronic form; it could be a card catalog at a library, your checkbook, a SQL Server database, an Excel spreadsheet, or even just a simple text file. Typically, the point of any database is to arrange data for ease and speed of search and retrieval—electronic or otherwise.

The database is the highest-level container that you will use to group all the objects and code that serve a common purpose. On an instance of the database server, you can have multiple databases, but best practices suggest using as few as possible for your needs. This container is often considered the level of consistency that is desired that all data is maintained at, but this can be overridden for certain purposes (one such case is that databases can be partially restored and be used to achieve quick recovery for highly available database systems.) A database is also where the storage on the file system meets the logical implementation. Until very late in this book, in Chapter 10, really, we will treat the database as a logical container and ignore the internal properties of how data is stored; we will treat storage and optimization primarily as a post-relational structure implementation consideration.

The next level of containership is the schema. You use schemas to group objects in the database with common themes or even common owners. All objects on the database server can be addressed by knowing the database they reside in and the schema, giving you what is known as the three-part name (note that you can set up linked servers and include a server name as well, for a four-part name):

```
databaseName.schemaName.objectName
```

Schemas will play a large part of your design, not only to segregate objects of like types but also because segregation into schemas allows you to control access to the data and restrict permissions, if necessary, to only a certain subset of the implemented database.

Once the database is actually implemented, it becomes the primary container used to hold, back up, and subsequently restore data when necessary. It does not limit you to accessing data within only that one database; however, it should generally be the goal to keep your data access needs to one database. In Chapter 9, we will discuss in some detail the security problems of managing security of data in separate databases.

■ **Note** The term "schema" has other common usages that you should realize: the entire structure for the databases is referred to as the schema, as are the Data Definition Language (DDL) statements that are used to create the objects in the database (such as CREATE TABLE and CREATE INDEX). Once we arrive to the point where we are talking about schema database objects, we will clearly make that delineation.

Understanding Tables, Rows, and Columns

The object that will be involved in almost all your designs and code is the table. The table is used to store information and will be used to represent some thing that you want to store data about. A table can be used to represent people, places, things, or ideas (i.e., nouns, generally speaking) about which information needs to be stored.

In a relational database, a table is a representation of data from which all the implementation aspects have been removed. It is basically data that has a light structure of having instances of some concept (like a person) and information about that concept (the person's name, address, etc). The instance is implemented as a row, and the information implemented in columns, which will be further defined in this section. A table is not to be thought of as having any order and should not be thought of as a location in some storage. As previously discussed in the "Taking a Brief Jaunt Through History" section of this chapter, one of the major design concepts behind a relational database system is that it is to be encapsulated from the physical implementation.

www.it-ebooks.info

A table is made up of rows of data, which are used to represent a single instance of the concept that the table represents. So if the table represents people, a row would represent a single person. Each row is broken up into columns that contain a single piece of information about whatever the row is representing. For example, the first name column of a row might contain "Fred" or "Alice".

"Atomic," or "scalar," which I briefly mentioned earlier, describes the type of data stored in a column. The meaning of "atomic" is pretty much the same as in physics. Atomic values will be broken up until they cannot be made smaller without losing the original characteristics. In chemistry molecules are made up of multiple atoms—H_2O can be broken down to two hydrogen atoms and one oxygen atom—but if you break the oxygen atom into smaller parts, you will no longer have oxygen (and you will probably find yourself scattered around the neighborhood along with parts of your neighbors).

A scalar value can mean a single value that is obvious to the common user, such as a single word or a number, or it can mean something like a whole chapter in a book stored in a binary or even a complex type, such as a point with longitude and latitude. The key is that the column represents a single value that resists being broken down to a lower level than what is needed when you start using the data. So, having a column that is defined as two independent values, say Column.X and Column.Y, is perfectly acceptable because they are not independent of one another, while defining a column to deal with values like 'Cindy,Leo,John' would likely be invalid, because that value would very likely need to be broken apart to be useful. However, if you will never need to programmatically access part of a value, it is, for all intents and purposes, a scalar value.

A very important concept of a table is that it should be thought of as having no order. Rows can be stored and used in any order, and columns needn't be in any fixed order either. This fundamental property will ideally steer your utilization of data to specify the output you need and to ask for data in a given order if you desire rows to be in some expected order.

Now, we come to the problem with the terms "table," "row," and "column." These terms are commonly used by tools like Excel, Word, and so on to mean a fixed structure for displaying data. For table, Dictionary.com (http://dictionary.reference.com) has the following definition for "table":

> *An orderly arrangement of data, especially one in which the data are arranged in columns and rows in an essentially rectangular form.*

When data is arranged in a rectangular form, it has an order and very specific locations. A basic example of this definition of "table" that most people are familiar with is a Microsoft Excel spreadsheet, such as the one shown in Figure 1-1.

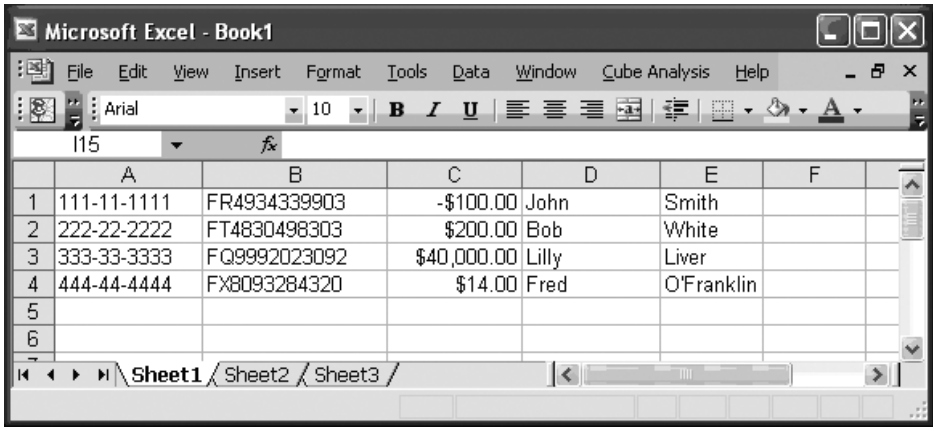


Figure 1-1. *Excel table*

www.it-ebooks.info

In Figure 1-1, the rows are numbered 1–6, and the columns are labeled A–F. The spreadsheet is a table of accounts. Every column represents some piece of information about an account: a Social Security number, an account number, an account balance, and the first and last names of the account holder. Each row of the spreadsheet represents one specific account. It is not uncommon to access data in a spreadsheet positionally (e.g., cell A1) or as a range of values (e.g., A1–B1) with no knowledge of the data's structure. As you will see, in relational databases, you access data not by its position but using values of the data themselves (this will be covered in more detail later in this chapter.)

In the next few tables, I will present the terminology for tables, rows, and columns and explain how they will be used in this book. Understanding this terminology is a lot more important than it might seem, as using these terms correctly will point you down the correct path for using relational objects. Let's look at the different terms and how they are presented from the following perspectives:

- *Relational theory*: This viewpoint is rather academic. It tends to be very stringent in its outlook on terminology and has names based on the mathematical origins of relational databases.
- *Logical/conceptual*: This set of terminology is used prior to the actual implementation phase.
- *Physical*: This set of terms is used for the implemented database. The word "physical" is bit misleading here, because the physical database is really an abstraction away from the tangible, physical architecture. However, the term has been ingrained in the minds of data architects for years and is unlikely to change.
- *Record manager*: Early database systems were involved a lot of storage knowledge; for example, you needed to know where to go fetch a row in a file. The terminology from these systems has spilled over into relational databases, because the concepts are quite similar.

Table 1-1 shows all of the names that the basic data representations (e.g., tables) are given from the various viewpoints. Each of these names has slightly different meanings, but are often used as exact synonyms.

■ **Note** The new datatypes, like XML, spatial types (geography and geography), hierarchyId, and even custom-defined CLR types, really start to muddy the waters of atomic, scalar, and nondecomposable column values. Each of these has some implementational value, but in your design, the initial goal is to use a scalar type first and one of the commonly referred to as "beyond relational" types as a fallback for implementing structures that are overly difficult using scalars only.

Next up, we look at columns. Table 1-2 lists all the names that columns are given from the various viewpoints, several of which we will use in the different contexts as we progress through the design process.

Finally, Table 1-3 describes the different ways to refer to a row.

If this is the first time you've seen the terms listed in Tables 1-1 through 1-3, I expect that at this point you're banging your head against something solid (and possibly wishing you could use my head instead) and trying to figure out why such a variety of terms are used to represent pretty much the same things. Many a flame war has erupted over the difference between a field and a column, for example. I personally cringe whenever a person uses the term "field," but I also realize that it isn't the worst thing if a person realizes everything about how a table should be dealt with in SQL but misuses a term.

www.it-ebooks.info

Table 1-1. *Breakdown of Basic Data Representation Terms*

Viewpoint	Name	Definition
Relational theory	Relation	This term is seldom used by nonacademics, but some literature uses it exclusively to mean what most programmers think of as a table. A relation consists of rows and columns, with no duplicate rows. There is absolutely no ordering implied in the structure of the relation, neither for rows nor for columns. *Note: Relational databases take their name from this term; the name does not come from the fact that tables can be related (relationships are covered later in this chapter).*
Logical/conceptual	Entity	An entity can be loosely represented by a table with columns and rows. An entity initially is not governed as strictly as a table. For example, if you are modeling a human resources application, an employee photo would be an attribute of the Employees entity. During the logical modeling phase, many entities will be identified, some of which will actually become tables and some will become several tables. The formation of the implementation tables is based on a process known as normalization, which we'll cover extensively in Chapter 6.
Physical	Recordset/rowset	A recordset, or rowset, is a tabular data stream that has been retrieved for use, such results sent to a client. Most commonly, it will be in the form of a tabular data stream that the user interfaces or middle-tier objects can use. Recordsets do have order, in that usually (based on implementation) the columns and the rows can be accessed by position and rows by their location in the table of data (however, accessing them in this way is questionable). Seldom will you deal with recordsets in the context of database design, but you will once you start writing SQL statements. A set, in relational theory terms, has no ordering, so technically a recordset is not a set *per se*.
Physical	Table	A table is almost the same as a relation. As mentioned, "table" is a particularly horrible name, because the structure that this list of terms is in is also referred to as a table. These structured lists, much like the Excel tables, have order. It cannot be reiterated enough that tables have *no* order (the "The Information Principle" section later in this chapter will clarify this concept further). The biggest difference between relations and tables is that tables technically may have duplicate rows (though they should not be allowed to). It is up to the developer to apply constraints to make certain that duplicate rows are not allowed. The term "tables" also has another common (though really not very correct) usage, in that the results of a query (including the intermediate results that occur as a query is processing multiple joins and the other clauses of a query) are also called tables, and the columns in these intermediate tables may not even have column names.
Record manager	File	In many nonrelational based database systems (such as Microsoft FoxPro), each operating system file represents a table (sometimes a table is actually referred to as a database, which is just way too confusing). Multiple files make up a database.

www.it-ebooks.info

Table 1-2. *Column Term Breakdown*

Viewpoint	Name	Definition
Logical/conceptual	Attribute	The term "attribute" is common in the programming world. It basically specifies some information about an object. In early logical modeling, this term can be applied to almost anything, and it may actually represent other entities. Just as with entities, normalization will change the shape of the attribute to a specific basic form.
Physical	Column	A column is a single piece of information describing what a row represents. Values that the column is designed to deal with should be at their lowest form and will not be divided for use in the relational language. The position of a column within a table must be unimportant to its usage, even though SQL does generally define a left-to-right order of columns in the catalog. All direct access to a column will be by name, not position(note that you can currently name the position of the column in the ORDER BY clause, but that is naming the position in the SELECT statement. Using the position in the ORDER BY clause is a bad practice however, and it is best to use one of the outputted names, including one of the aliases).
Recordmanager	Field	The term "field" has a couple of meanings. One meaning is the intersection of a row and a column, as in a spreadsheet (this might also be called a cell). The other meaning is more related to early database technology: a field was the offset location in a record, which as I will define in Table 1-3, is a location in a file on disk. There are no set requirements that a field store only scalar values, merely that it is accessible by a programming language.

Table 1-3. *Row Term Breakdown*

Viewpoint	Name	Definition
Relationaltheory	Tuple	A tuple (pronounced "tupple," not "toople") is a finite set of related named value pairs. By "named," I mean that each of the values is known by a name (e.g., Name: Fred; Occupation: gravel worker). "Tuple" is a term seldom used in a relational context except in academic circles, but you should know it, just in case you encounter it when you are surfing the Web looking for database information. In addition, this knowledge will make you more attractive to the opposite sex—if only. Note that tuple is used in cubes and MDX to mean pretty much the same concept. Ultimately, "tuple" is a better term than "row," since a row gives the impression of something physical, and it is essential to not think this way when working in SQL Server with data.
Logical/ conceptual	Instance	Basically, this is one of whatever is being represented by the entity. This term is far more commonly used by object oriented programmers to represent a instance of an object.
Physical	Row	A row is essentially the same as a tuple, though the term "row" implies it is part of something (in this case, a row in a table). Each column represents one piece of data of the thing that the row has been modeled to represent.
RecordManager	Record	A record is considered to be a location in a file on disk. Each record consists of fields, which all have physical locations. This term should not be used interchangeably with the term "row." A row has no physical location, just data in columns.

www.it-ebooks.info

Working with Missing Values (NULLs)

In the previous section, we noted that columns are used to store a single value. The problem with this is that often you will want to store a value, but at some point in the process, you may not know the value. As mentioned earlier, Codd's third rule defined the concept of NULL values, which was different from an empty character string or a string of blank characters or zero used for representing missing information in a systematic way, independent of data type. All datatypes are able to represent a NULL, so any column may have the ability to represent that data is missing.

When representing missing values, it is important to understand what the value represents. Since the value is missing, it is assumed that there exists a value (even if that value is that there is no value.) Because of this, no two values of NULL are considered to be equal, and you have to treat the value like it could be any value at all. This brings up a few interesting properties of NULL that make it a pain to use, though it is very often needed:

- Any value concatenated with NULL is NULL. NULL can represent any valid value, so if an unknown value is concatenated with a known value, the result is still an unknown value.
- All math operations with NULL will return NULL, for the very same reason that any value concatenated with NULL returns NULL.
- Logical comparisons can get tricky when NULL is introduced because NULL <> NULL (this comparison actually is NULL, not FALSE, since any unknown value might be equal to another unknown value, so it is unknown if they are not equal).

Let's expand this last point somewhat. When NULL is introduced into Boolean expressions, the truth tables get more complex. Instead of a simple two-condition Boolean value, when evaluating a condition with NULLs involved, there are three possible outcomes: TRUE, FALSE, or UNKNOWN. Only if the search condition evaluates to TRUE will a row appear in the results. As an example, if one of your conditions is NULL = 1, you might be tempted to assume that the answer to this is FALSE, when in fact this actually resolves to UNKNOWN.

This is most interesting because of queries such as the following:

```
SELECT CASE WHEN 1=NULL or NOT(1=NULL) THEN 'True' ELSE 'NotTrue' END
```

Many people would expect NOT(1 = NULL) to evaluate to TRUE, but in fact, 1 = NULL is UNKNOWN, and NOT(UNKNOWN) is also UNKNOWN. The opposite of unknown is not, as you might guess, known. Instead, since you aren't sure if UNKNOWN represents TRUE or FALSE, the opposite might also be TRUE or FALSE.

Table 1-4 shows the truth table for the NOT operator.

Table 1-5 shows the truth tables for the AND and OR operators.

Table 1-4. *NOT Truth Table*

Operand	NOT(Operand)
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

Table 1-5. *AND and OR Truth Table*

Operand1	Operand2	Operand1 AND Operand2	Operand1 OR Operand2
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	UNKNOWN	FALSE	UNKNOWN

www.it-ebooks.info

In this introductory chapter, my main goal is to point out that NULLs exist and are part of the basic foundation of relational databases (along with giving you a basic understanding of why they can be troublesome); I don't intend to go too far into how to program with them. The goal in your designs will be to minimize the use of NULLs, but unfortunately, completely ignoring them is impossible, particularly because they begin to appear in your SQL statements even when you do an outer join operation.

Though using NULLs to represent missing values seems simple enough, often a designer will try to choose a value outside of a columns domain to denote this value. (This value is sometimes referred to as a *sentinel* value, the domain of the column represents legitimate values and will be discussed in the next section) For decades, programmers have used ancient dates in a date column to indicate that a certain value does not matter, a negative value where it does not make sense in the context of a column for a missing number value, or simply a text string of 'UNKNOWN' or 'N/A.' These approaches seem fine on the surface, but in the end, special coding is still required to deal with these values, and the value truly must be illegal, for all uses other than missing data. For example, using a string value of 'UNKNOWN' could be handled as follows:

```
IF (value<>'UNKNOWN') THEN …
```

But what happens if the user needs to put actually use the value 'UNKNOWN' as a piece of data ? Now you have to find a new stand-in for NULL and go back and change all of the code, and that is a pain. You have eliminated one troublesome but well-known problem of dealing with three-value logic and replaced it with a problem that now requires all of your code to be written using a nonstandard pattern.

What makes this implementation using a stand-in value to represent NULL more difficult than simply sticking to NULL is that it is not handled the same way for all types, or even the same way for the same type every time. Special coding is needed *every time* a new type of column is added, and every programmer and user must know the conventions. Instead, use a value of NULL, which in relational theory means an empty set or a set with no value.

Defining Domains

As we start to define the columns of tables, it becomes immediately important to consider what types of values we want to be able to store. For each column we will define a *domain* as the set of valid values that the column is allowed to store. As you define the domain, the concepts of implementing a physical database aren't really important; some parts of the domain definition may just end up just using them as warnings to the user. For example, consider the following list of possible types of domains that you might need to apply to a date type column you have specified to form a domain for an EmployeeDateOfBirth column:

- The value must be a calendar date with no time value.
- The value must be a date prior to the current date (a date in the future would mean the person has not been born).
- The date value should evaluate such that the person is at least 16 years old, since you couldn't legally hire a 10-year-old, for example.
- The date value should be less than 70 years ago, since rarely will an employee (especially a new employee) be that age.
- The value must be less than 120 years ago, since we certainly won't have a new employee that old. Any value outside these bounds would clearly be in error.

Starting with Chapter 6, we'll cover how you might implement this domain, but during the design phase, you just need to document it. The most important thing to note here is that not all of these rules are expressed as 100% required. For example, consider the statement that the date value should be less than 120 years ago. During your early design phase, it is best to define everything about your domains (and really everything you find out about, so it can be implemented in some manner, even if it is just a message box asking the user "Really?" for values out of normal bounds).

www.it-ebooks.info

As you start to create your first model, you will find a lot of commonality among attributes. As you start to build your second model, you will realize that you have done a lot of this before. After 100 models, trying to decide how long to make a person's first name in a customer table will be tedious. To make this process easier and to achieve some standards among your models, a great practice (not just a best practice!) is to give common domain types names so you can associate them to attributes with common needs. For example, you could define the type we described at the start of this section as an employeeBirthDate domain. Every time an employee birth date is needed, it will be associated with this named domain. Admittedly, using a named domain is not always possible, particularly if you don't have a tool that will help you manage it, but the ability to create reusable domain types is definitely something I look for in a data modeling tool.

Domains do not have to be very specific, because often we just use the same kinds of value the same way. For example, if we have a count of the number of puppies, that data might resemble a count of handguns. Puppies and handguns don't mix (only full grown dogs should use firearms!). For example, you might have the following named domains:

- positiveInteger: Integer values 1 and greater
- date: Any valid date value (with no time of the day value)
- emailAddress: A string value that must be formatted as a valid e-mail address
- 30CharacterString: A string of characters that can be no longer than 30 characters

Keep in mind that if you actually define the domain of a string to any positive integer, the maximum is theoretically infinity. Today's hardware boundaries allow some pretty far out maximum values (e.g., 2,147,483,647 for a regular integer, and a *very* large number for a bigint type). It is fairly rare that a user will have to enter a value approaching 2 billion, but if you do not constrain the data within your domains, reports and programs will need to be able handle such large data. I will cover this more in Chapter 7 when we discuss data integrity as well as Chapter 8, when I will discuss patterns of implementation to meet requirements.

■ **Note** Domains and columns need not contain only single scalar values. As long as the values are accessible only through predefined operations, you can have fixed vector values, such as a point in a plane (e.g., longitude and latitude). starting with SQL Server 2008, we have spatial datatypes that represent a scalar (a point or a shape), but the internals can allow a nonfixed number of points to form complex shapes. I won't specifically be covering spatial types in this book.

Storing Metadata

Metadata is data stored to describe other data. Knowing how to find information about the data stored in your system is a very important aspect of the documentation process. As previously mentioned in this chapter, Codd's fourth rule states that "the database description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to regular data." This means you should be able to interrogate the system metadata using the same language you use to interrogate the user data (i.e., SQL).

According to relational theory, a relation consists of two parts:

- *Heading*: The set of column name/datatype name pairs that define the columns of the table
- *Body*: The rows that make up the table

www.it-ebooks.info

In SQL Server—and most databases—it is common to consider the catalog as a collective description of the heading of tables and other structures in the database. SQL Server exposes the heading information in a couple of ways:

- *In a set of views known as the information schema*: Use this as the primary means of viewing the properties of the objects in your database as far as possible. It consists of a standard set of views used to view the system metadata and should exist on all database servers of any brand.
- *In the SQL Server–specific catalog (or system) views*: These views give you information about the implementation of your objects and many more properties of your system.

It is a very good practice to maintain a greater amount of metadata about your databases to further define a table's or column's purpose. This is commonly done in spreadsheets and data modeling tools, as well as using custom metadata storage built into the RDBMS (e.g., extended properties in SQL Server.)

Assigning Uniqueness Constraints (Keys)

In relational theory, a relation, by definition, cannot represent duplicate tuples. In RDBMS products, however, no enforced limitation says that there must not be duplicate rows in a table. However, it is the considered recommendation of myself and most data architects that all tables have at least one defined uniqueness criteria to fulfill the mandate that rows in a table are accessible by values in the table. Unless each row is unique from all other rows, there would be no way to retrieve a single row.

To define the uniqueness criteria, we will define keys. Keys define uniqueness for a table over one or more columns that will then be guaranteed as having distinct values from all other rows in the table. Generically, a key is usually referred to as a candidate key, because you can have more than one key defined for a table, and a key may play a few roles (primary or alternate) for a table, as will be discussed later in this section.

As already mentioned (and will be reiterated frequently throughout the book), every table should have at least one candidate key—an attribute (or combination of attributes) that can uniquely and unambiguously identify each instance of an entity (or, in the physical model, each row in the table). Consider the following table, T, with columns X and Y:

```
X   Y
--- ---
1   1
2   1
```

If you attempted to add a new row with values X:1, Y:1, there would then be two identical rows in the table. If this were allowed, it would be problematic for a couple of reasons:

- Because rows in a table are unordered, without keys, there would be no way to tell which of the rows with value X:1, Y:1 in the preceding table was which. Hence, it would be impossible to distinguish between these rows, meaning that there would be no logical method of accessing a single row. This using, changing, or deleting an individual row difficult without resorting to tricks that Microsoft has allowed in SQL Server (such as using the TOP operator in statements).
- If more than one row has the same values, it describes the same object, so if you try to change one of the rows, the other row must also change, which becomes a messy situation.

If we define a key on column X, the previous attempt to create a new row would fail, as would any other insert of a value of 1 for the X column, such as X:1, Y:3. Alternatively, if you define the key using both columns X and Y (known as a composite key, i.e., a key that has more than one column), the X:1 Y:3 creation would be allowed, but attempting to create a row where X:1 Y:1 is inserted would still be forbidden.

www.it-ebooks.info

■ **Note** In a practical sense, no two rows can actually be the same, because there are realities of the implementation, such as the location in the storage system where the rows are stored. However, this sort of thinking has no place in relational database design, where it is our goal to largely ignore the storage aspects of the implementation.

So what is the big deal? If you have two rows with the same values for X and Y, what does it matter? Consider a table that has three columns:

```
MascotName MascotSchool PersonName
---------- ------------ -----------
Smokey     UT           Bob
Smokey     UT           Fred
```

Now, you want to answer the question of who plays the part of Smokey for UT. Assuming that there is only one actual person who plays the part, you retrieve one row. Since we have stated that tables are unordered, you could get either row, and hence either person. Applying a candidate key to MascotName and MascotSchool will ensure that a fetch to the table to get the mascot named Smokey that cheers for UT will get the name of only one person. (Note that this example is an oversimplification of the overall problem, since you may or may not want to allow multiple people to play the part for a variety of reasons. But we are defining a domain in which only one row should meet the criteria). Failure to identify the keys for a table is one of the largest blunders that a designer will make, mostly because during early testing, it won't be recognized as testers tend to test like a good user to start with (usually, they're programmers testing their own code). That plus the fact that testing is often the first cost to be cut when time is running out for release means that major blunders can occur.

In summary, a candidate key (or simply "key" for short) defines the uniqueness of rows over a column or set of columns. A table may have multiple keys to maintain the uniqueness of its rows, and a key may have as many columns as is needed to define its uniqueness.

Types of Keys

Two types of keys are defined: primary and alternate (you may have also heard the term "foreign key," but this is a reference to a key and will be defined later in this chapter in the "Understanding Relationships" section.) A primary key (PK) is used as the primary identifier for an entity. It is used to uniquely identify every instance of that entity. If you have more than one key that can perform this role, after the primary key is chosen, each remaining candidate key would be referred to as an alternate key (AK). There is technically no difference in implementation of the two (though a primary key, by definition, will not allow nullable columns, as this ensures that at least one known key value will be available to fetch a row from the unordered set. Alternate keys, by definition, do allow NULL values that are treated as different values in most RDBMSs (which would follow along with the definition of NULL presented earlier), but in SQL Server, a unique constraint (and unique index) will treat all NULL values as the same value and only a single instance of NULL may exist. In Chapter 7, we will discuss implementation patterns for implementing uniqueness conditions of several types in more detail during design, and again in Chapter 8, we will revisit the methods of implementing the different sorts of uniqueness criteria that exist.

As example keys, in the United States, the Social Security Number/Work Visa Number/Permanent Resident Number are unique for all people (some people have more than one of these, but no legitimate duplication is recognized. Hence, you wouldn't want two employees with the same Social Security number, unless you are trying to check "IRS agent" off your list of people you haven't had a visit from). However, Social Security number is not a good key to share because of security risks, so every employee probably also has a unique, company-supplied identification number. One of these could be chosen as a PK (most likely the employee number), and the other would then be an AK.

www.it-ebooks.info

The choice of primary key is largely a matter of convenience and ease of use. We'll discuss primary keys later in this chapter in the context of relationships. The important thing to remember is that when you have values that should exist only once in the database, you need to protect against duplicates.

Choosing Keys

While keys can consist of any number of columns, it is best to limit the number of columns in a key as much as possible. For example, you may have a Book table with the columns Publisher_Name, Publisher_City, ISBN_Number, Book_Name, and Edition. From these attributes, the following three keys might be defined:

- Publisher_Name, Book_Name, Edition: A publisher will likely publish more than one book. Also, it is safe to assume that book names are not unique across all books. However, it is probably true that the same publisher will not publish two books with the same title and the same edition (at least, we can assume that this is true!).
- ISBN_Number: The ISBN number is the unique identification number assigned to a book when it is published.
- Publisher_City, ISBN_Number: Because ISBN_Number is unique, it follows that Publisher_City and ISBN_Number combined is also unique.

The choice of (Publisher_Name, Book_Name) as a composite candidate key seems valid, but the (Publisher_City, ISBN_Number) key requires more thought. The implication of this key is that in every city, ISBN_Number can be used again, a conclusion that is obviously not appropriate. This is a common problem with composite keys, which are often not thought out properly. In this case, you might choose ISBN_Number as the PK and (Publisher_Name, Book_Name) as the AK.

■ **Note** Do not confuse unique indexes with keys. There may be valid performance-based reasons to implement the Publisher_City, ISBN_Number index in your SQL Server database. However, this would not be identified as a key of a table. In Chapter 6, we'll discuss implementing keys, and in Chapter 10, we'll cover implementing indexes for data access enhancement.

Having established what keys are, we'll next discuss the two main types of keys:

- *Natural keys*: The values that make up the key have some recognizable connection to the row data
- *Surrogate keys*: The value has no connection to the row data but is simply used as a stand-in for the natural key for complexity or performance reasons

Natural Keys

Natural keys are generally some real attribute of an entity that logically, uniquely identify each instance of an entity. From our previous examples, all of our candidate keys so far—employee number, Social Security number (SSN), ISBN, and the (Publisher_Name, Book_Name) composite key—have been examples of natural keys. These are values that a user would recognize and should always be values that are presented to the users.

www.it-ebooks.info

Some common examples of good natural keys follow:

- *For people*: Driver's license numbers (including the state of issue), company identification number, or other assigned IDs (e.g., customer numbers or employee numbers)
- *For transactional documents* (e.g., invoices, bills, and computer-generated notices): Usually assigned some sort of number when they are created
- *For products for sale*: Product numbers (product names are likely not unique)
- *For companies that clients deal with*: Commonly assigned a customer/client number for tracking
- *For buildings*: A complete street address, including the postal code
- *For mail*: The addressee's name and address and the date the item was sent

Be careful when choosing a natural key. Ideally, you are looking for something that is stable, that you can control, and that is definitely going to allow you to uniquely identify every row in your database.

One thing of interest here is that what might be considered a natural key in your database is often not actually a natural key in the place where it is defined, for example, the driver's license number of a person. In the example, this is a number that every person has (or may need before inclusion in our database). However, the value of the driver's license number is just a series of integers. This number did not appear tattooed on the back of the person's neck at birth. In the database where that number was created, it was actually more of a surrogate key (which we will define in a later section).

Values for which you cannot guarantee uniqueness, no matter how unlikely the case, make poor keys. Given that three-part names are common in the United States, it is usually relatively rare that you'll have two people working in the same company or attending the same school who have the same three names. (Of course, as the number of people who work in the company increases, the odds will go up that you will have duplicates.) If you include prefixes and suffixes, it gets even more unlikely, but "rare" or even "extremely rare" cannot be implemented in a manner that makes a reasonable key. If you happen to hire two people called Sir Lester James Fredingston III (and wouldn't that be fun!), the second of them probably isn't going to take kindly to being called Les for short just so your database system can store his name (and a user would, in fact, do that).

One notable profession where names must be unique is acting. No two actors who have their union cards can have the same name. Some change their names from Archibald Leach to something more pleasant like Cary Grant, but in some cases, the person wants to keep his or her name, so in the actors database, the Screen Actors' Guild adds a uniquifier to the name to make it unique. A uniquifier is a nonsensical value (like a sequence number) that is added to nonunique values to produce uniqueness where it is required for a situation like this where names are very important to be dealt with as unique.

For example, six people (up from five in the last edition, just to prove I am diligent in giving you the most up-to-date information by golly) are listed on the Internet Movie Database site (http://www.imdb.com) with the name Gary Grant (not Cary, but Gary). Each has a different number associated with his name to make him a unique Gary Grant. (Of course, none of these people have hit the big time yet, but watch out—it could be happening soon!)

■ **Tip** We tend to think of names in most systems as a kind of semiunique natural key. This isn't good enough for identifying a single row, but it's great for a human to find a value. The phone book is a good example of this. Say you need to find Ray Janakowski in the phone book. There might be more than one person with this name, but it might be a good enough way to look up a person's phone number. This semiuniqueness is a very interesting attribute of a table and should be documented for later use, but only in rare cases would you make a key from semiunique values using uniquifiers. In Chapter 8, we will cover the process of defining and implementing this case, which I refer to as

www.it-ebooks.info

"likely uniqueness." Likely uniqueness criteria basically states that you should ask for verification if you try to create to people with the same or extremely similar names. Finding and dealing with duplicate data is a lot harder once the data is stored.

Smart Keys

A commonly occurring type of natural key in computer systems is a smart, or intelligent, key. Some identifiers will have additional information embedded in them, often as an easy way to build a unique value for helping a human identify some real-world thing. In most cases, the smart key can be disassembled into its parts. In some cases, however, the data will probably not jump out at you. Take the following example of the fictitious product serial number XJV102329392000123:

- *X*: Type of product (LCD television)
- *JV*: Subtype of product (32-inch console)
- *1023*: Lot that the product was produced in (batch number 1023)
- *293*: Day of year
- *9*: Last digit of year
- *2*: Original Color
- *000123*: Order of production

The simple-to-use smart key values serve an important purpose to the end user; the technician who received the product can decipher the value and see that, in fact, this product was built in a lot that contained defective whatchamajiggers, and he needs to replace it. The essential thing for us during the logical design phase is to find all the bits of information that make up the smart keys because each of these values is likely going to need to be stored in its own column.

Smart keys, while useful in some cases, often present the database implementor with problems that will occur over time. When at all possible, instead of implementing a single column with all of these values, consider having multiple column values for each of the different pieces of information and calculating the value of the smart key. The end users get what they need, and you, in turn, get what you need—a column value that never needs to be broken down into parts to work with.

A couple of big problems with smart keys are that you could run out of unique values for the constituent parts or some part of the key (e.g., the product type or subtype) may change. Being very careful and planning ahead well are imperative if you use smart keys to represent multiple pieces of information. When you have to change the format of smart keys, making sure that different values of the smart key are actually valid becomes a large validation problem. Note, too, that the color position can't indicate the current color, just the original color. This is common with automobiles that have been painted: the VIN number includes color, but the color can change.

■ **Note** Smart keys are useful tools to communicate a lot of information to the user in a small package. However, all the bits of information that make up the smart key need to be identified, documented, and implemented in a straightforward manner. Optimum SQL code expects the data to all be stored in individual columns, and as such, it is of great importance that you needn't ever base computing decisions on decoding the value. We will talk more about the subject of choosing implementation keys in Chapter 6.

---

www.it-ebooks.info

Surrogate Keys

Surrogate keys (sometimes called artificial keys) are kind of the opposite of natural keys. The word surrogate means "something that substitutes for," and in this case, a surrogate key serves as a substitute for a natural key. Sometimes, you may have no natural key that you think is stable or reliable enough to use, in which case you may decide to use a surrogate key. In reality, many of our examples of natural keys were technically surrogate keys in their original database but were elevated to a natural status by usage in the real world.

A surrogate key can uniquely identify a row in a table, but it has no actual meaning with regard to that table other than to represent existence. Surrogate keys are usually manufactured by the system as a convenience to either the RDBMS or the client. Common methods for creating surrogate key values are using a monotonically increasing number (e.g., an Identity column), some form of hash function, or even a globally unique identifier (GUID), which is a very long (16-byte) identifier that is unique on all machines in the world. Note that being computer generated doesn't make a key a surrogate; you will generate all sorts of keys that will be treated as natural keys, like a customer ID.

The concept of a surrogate key can be troubling to purists and may start an argument or two. Since the surrogate key does not describe the row at all, can it really be an attribute of the row? The question is valid, but surrogate keys have a number of nice values for usage that make implementation so easy. For example, an exceptionally nice aspect of a surrogate key is that the value of the key should never change. This, coupled with the fact that surrogate keys are always a single column, makes several aspects of implementation far easier than they otherwise might be.

Usually, a true surrogate key is never shared with any users. It will be a value generated on the computer system that is hidden from use, while the user directly accesses only the natural keys' values. Probably the best reason for this limitation is that once a user has access to a value, it may need to be modified. For example, if you were customer 0000013 or customer 00000666, you might request a change.

Just as the driver's license number probably has no meaning to the police officer other than a means to quickly check your records (though an article on highprogrammer.com shows that, in some states, this is not the case), the surrogate is used to make working with the data programmatically easier. Since the source of the value for the surrogate key does not have any correspondence to something a user might care about, once a value has been associated with a row, there is not ever a reason to change the value. This is an exceptionally nice aspect of surrogate keys. The fact that the value of the key does not change, coupled with the fact that it is always a single column, makes several aspects of implementation far easier. This will be made clearer later in this book when we cover choosing a primary key.

Thinking back to the driver's license analogy, if the driver's license has just a single value (the surrogate key) on it, how would Officer Uberter Sloudoun determine whether you were actually the person identified? He couldn't, so there are other attributes listed, such as name, birth date, and usually your picture, which is an excellent unique key for a human to deal with (except possibly for identical twins, of course). In this very same way, a table ought to have other keys defined as well, or it is not a proper table.

Consider the earlier example of a product identifier consisting of seven parts:

- *X*: Type of product (LCD television)
- *JV*: Subtype of product (32-inch console)
- *1023*: Lot that the product was produced in (batch 1023)
- *293*: Day of year
- *9*: Last digit of year
- *2*: Original color
- *000123*: Order of production

www.it-ebooks.info

A natural key would consist of these seven parts. There is also a product serial number, which is the concatenation of the values such as XJV102329392000123, to identify the row. Say you also have a surrogate key column value in the table with a value of 10. If the only key defined on the rows is the surrogate, the following situation might occur if the same data is inserted other than the surrogate (which gets an automatically generated value of 3384):

```
SurrogateKey   ProductSerialNumber ProductType ProductSubType Lot  Date     ColorCode   …
-------------- ------------------- ----------- -------------- ---- -------- --------
10             XJV102329392000123  X           JV             1023 20091020 2           …
3384           XJV102329392000123  X           JV             1023 20091020 2           …
```

The two rows are not technically duplicates, but since the surrogate key values have no real meaning, in essence these are duplicate rows, since the user could not effectively tell them apart. This situation gets very troublesome when you start to work with relationships (which we cover in more detail later in this chapter). The value 10 and 3384 are stored in other tables as references to this table, so it looks like two different products are being referenced when in reality there is only one.

This sort of problem is common, because most people using surrogate keys do not understand that a surrogate is a stand-in for a natural key, and only having a surrogate key opens you up to having rows with duplicate data in the columns where data has some logical relationship. A user looking at the preceding table would have no clue which row actually represented the product he or she was after, or if both rows did.

■ **Note** When doing early design, I tend to model each table with a surrogate primary key, since during the design process I may not yet know what the final keys will turn out to be. In systems where the desired implementation does not include surrogates, the process of designing the system will eliminate the surrogates. This approach will become obvious throughout this book, starting with the conceptual model in Chapter 4.

Understanding Relationships

In the previous section, we established what a table is and how tables are to be structured (especially with an eye on the future tables you will create), but a table by itself can be a bit boring. To make tables more interesting, and especially to achieve some of the structural requirements to implement tables in the desired shapes, you will need to link them together (sometimes even linking a table to itself). You do this by recognizing and defining the relationship between the tables. Without the concept of a relationship, it would be necessary to simply put all data into a single table when data was related to itself, which would be a very bad idea because of the need to repeat data over and over (repeating groups of data is the primary no-no in good database design).

A term that we need to establish is one that you no doubt will already have heard as a reader of this book: foreign key. A foreign key is used to establish a link between two tables by stating that a set of column values in one table is required to match the column values in a candidate key in another (commonly the primary key but any declared candidate key will do).

The foreign key is one of the most important tools to maintaining the integrity of the database, but a common mistake when discussing relationships is to think that all relationships between tables directly correspond to a foreign key. During the design phase, this is often not going to be the case. Sometimes, additional tables will need to be created to implement a relationship, and sometimes, you will not be able to implement a relationship using simple SQL constructs.

www.it-ebooks.info

When defining the relationship of one entity to another, several factors are important:

- *Involved tables*: The tables that are involved in the relationship will be important to how easy the relationship is to work with. In the reality of defining relationships, the number of related tables need not be two. Sometimes, it is just one, such as an employee table where you need to denote that one employee works for another, or sometimes, it is more than two; for example, Book Wholesalers, Books, and Book Stores are all commonly tables that would be related to one another in a complex relationship.
- *Ownership*: It is common that one table will own the other table. For example, an invoice will own the invoice line items. Without the invoice, there would be no line items.
- *Cardinality*: Cardinality indicates the number of instances of one entity that can be related to another. For example, a person might be allowed to have only one spouse (would you really want more?), but a person could have any number of children (still, I thought one was a good number there too!).

In every relationship between tables that we can implement in SQL, there will be two tables. The relationship is established by taking the primary key columns and placing them in a different table (sometimes referred to as "migrating the column"). The table that provides the key that is migrated is referred to as the parent in the relationship, and the one receiving the migrated key is the child.

For an example of a relationship between two tables, consider the relationship between a Parent table, which stores the SSNs and names of parents, and a Child table, which does the same for the children, as shown in Figure 1-2. Bear in mind a few things. First, this is a simple example that does not take into full consideration all of the intricacies of people's names. Second, the parent and child might be located in the same table; this will be discussed later in this book.

Parent

Parent SSN	Parent Name
111-11-1111	Larry Bull
222-22-2222	Fred Badezine

Child

Child SSN	Child Name	Parent SSN
333-33-3333	Tay	111-11-1111
444-44-4444	Maya	222-22-2222
555-55-5555	Riely	222-22-2222

Figure 1-2. *Sample parent and child tables*

In the Child table, the Parent SSN is the foreign key (denoted in these little diagrams using a double line). It is used in a Child row to associate the child with the parent. From these tables, you can see that Tay's dad is Larry Bull, and the parent of Maya is Fred Badezine (oh, the puns!).

■ **Note** As a reminder, the integrity independence rule (Codd's twelfth rule) requires that for all nonoptional foreign key values in the database, there must be a matching primary key value in the related table.

Cardinality is the next question. The realities of the relationship between parent and child dictate that

- One parent can have any number of children, even zero (of course, in the real world these would be more realistically referred to as potential parents).

www.it-ebooks.info

- Depending on the purpose of the database, the child can have a limited number of parents: a fixed number of two if the database stores biological parents and zero or more if you are talking about guardians, living parents, and so on in the same table.

We will cover how to implement the relationships in later chapters, particularly Chapter 7, where we will start to cover design patterns. Relationships can be divided at this point into two basic types based on the number of tables that are involved in the relationship:

- *Binary relationships*: Those between two tables
- *Nonbinary relationships*: Those between more than two tables

The biggest difference between the two types of relationships is that the binary relationship is very straightforward to implement using foreign keys, as we have discussed previously. When more than two tables are involved, all tables cannot necessarily be implemented in SQL in a natural manner, but they still do exist and will needs solutions (often these will seem clever, and they are, but they will generally be very common and well known in the industry, such as fixes involving multiple tables each related to one another with binary relationships).

When you are doing your early design, you need to keep this distinction in mind and learn to recognize each of the possible relationships. When I introduce data modeling in Chapter 2, you'll learn how to represent each of these in a data model.

Working with Binary Relationships

The number of rows that may participate in each side of the relationship is known as the cardinality of the relationship. Different cardinalities of binary relationships will be introduced in this section:

- One-to-many relationships
- Many-to-many relationships

Each of these relationship types and their different subtypes has specific uses and specific associated challenges.

One-to-Many Relationships

One-to-many relationships are the class of relationships whereby one table migrates its primary key to another table as a foreign key. As discussed earlier, this is commonly referred to as a parent/child relationship and concerns itself only with the relationship between exactly two tables. A child may have at most, one parent, but a parent may have many children. The generic name of parent child relationships is one-to-many, but when implementing the relationship, a more specific specification of cardinality is very common, where the one part of the name really can mean zero or one (but never greater than one, as that will be a different type called a many-to-many relationship), and "many" can mean zero, one, a specific amount, or an unlimited amount.

It should be immediately clear what when the type of relationship starts with "one," as in "one-to . . ." one row is related to some number of other rows. However, sometimes a child row can be related to zero parent rows. This case is often referred to as an optional relationship. If you consider the earlier Parent/Child example, if this relationship were optional, a child may exist without a parent. If the relationship between parent and child were optional, it would be OK to have a child named Sonny who did not have a parent (well, as far as the database knows, Sonny had a wonderful childhood without a parent), as shown in Figure 1-3.

The missing value would be denoted by NULL, so the row for Sonny would be stored as ChildSSN:'666-66-6666', ChildName:'Sonny', ParentSSN:NULL. For the general case, we (and most others in normal conversation) will speak in terms of one-to many relationships, just for ease of discussion. However,

www.it-ebooks.info

Parent

Parent SSN	Parent Name
111-11-1111	Larry Bull
222-22-2222	Fred Badezine

Child

Child SSN	Child Name	Parent SSN
333-33-3333	Tay	111-11-1111
444-44-4444	Maya	222-22-2222
555-55-5555	Riely	222-22-2222
666-66-6666	Sonny	

Figure 1-3. *Sample table including a parentless child*

in more technical terms, there are several different variations of the one-to-(blank) theme that have different implications later during implementation that we will cover in this section:

- *One-to-many*: This is the general case, where "many" can be between zero and infinity.
- *One–to–exactly N*: In this case, one parent row is required to be related to a given number of child rows. For example, a child can have only two biological parents. The common case is one-to-zero or one-to-one.
- *One–to–between X and Y*: Usually, the case that X is 0 and Y is some boundary set up to make life easier. For example, a user may have a maximum of two usernames.

One-to-Many (The General Case)

The one-to-many relationship is the most common and most important relationship type. For each parent row, there may exist unlimited child rows. An example one-to-many relationship might be Customer to Orders, as illustrated in Figure 1-4.

Customer Number	Name
1	Joe's Fisheteria
2	Betty's Bass Shop
3	Fred Fish

Customer Number	Order Number	Order Date
1	100000002	20070510
1	100000004	20070511
2	100000008	20070522
2	100000009	20070701
2	100000022	20070531
3	100000028	20070613

Figure 1-4. *One-to-many example*

Quite often, a one-to-many relationship will implement a relationship between two tables that indicates that the parent has (or "has a") child, a team has players; a class has students. This category generally indicates ownership by the parent of the child. "Has" relationships often indicate an attribute of the parent that has many values.

www.it-ebooks.info

A special type of one-to-many relationship is a recursive relationship. In a recursive relationship, the parent and the child are from the same table, and more often than not, the relationship is set up as a single table. This kind of relationship is used to model a tree data structure using SQL constructs. As an example, consider the classic example of a bill of materials. Take something as simple as a ceiling fan. In and of itself, a ceiling fan can be considered a part for sale by a manufacturer, and each of its components is, in turn, also a part that has a different part number. Some of these components also consist of parts. In this example, the ceiling fan could be regarded as made up recursively of each of its parts, and in turn, each part consists of all of its constituent parts.

The following table is small subset of the parts that make up a ceiling fan. Parts 2, 3, and 4 are all parts of a ceiling fan. You have a set of blades and a light assembly (among other things). Part 4, the globe that protects the light, is part of the light assembly. (OK, I had better stop here, or some of you might go to The Home Depot rather than read the rest of this book!)

```
Part Number   Description          Used in Part Number
------------- -------------------- -------------------
1             Ceiling Fan          NULL
2             White Fan Blade Kit  1
3             Light Assembly       1
4             Light Globe          3
5             White Fan Blade      2
```

To read this data, you would start at Part Number 1, and you can see what parts make up that part, which is a fan blade kit and a light assembly. Now, you have the parts with number 2 and 3, and you can look for parts that make them up, which gets you part number 4 and 5. (Note that the algorithm we just used is known as a breadth-first search, where you get all of the items on a level in each pass though the data. Not terribly important at this point, but it will come up in Chapter 8 when we are discussing design patterns.)

One–to–Exactly N Relationship

Often, some limit to the number of children is required by the situation being modeled or a business rule. For example, a business rule might state that a user must have two e-mail addresses. Figure 1-5 shows an example of one-to-two relationship cardinality. It's not particularly a likely occurrence to have names like this, but the points here are that each relationship has one parent and exactly two children and that examples of this type are pretty rare.

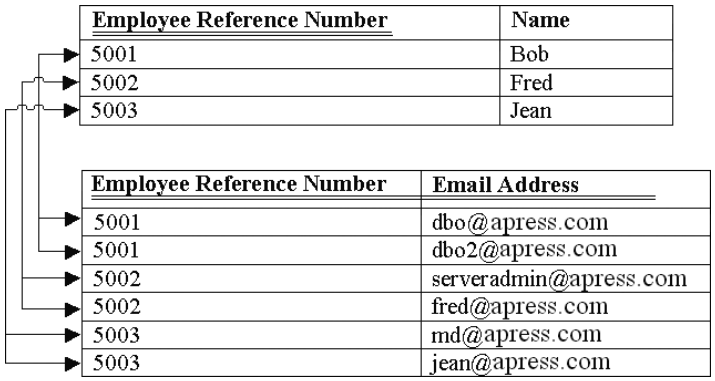


Figure 1-5. *Example of a one-to-two relationship*

www.it-ebooks.info

The most typical version of a one–to–exactly N relationship type that gets used is a one-to-one relationship. This indicates that for any given parent, there may exist exactly one instance of the child. Another example here is that a child has only two biological parents, but it gets trickier going the other direction, because parents have no logical limitation on the number of children they can have (other than the amount of sleep they need, that is).

A one-to-one relationship may be a simple "has a" relationship (i.e., a house has a location), or it may be what is referred to as an "is a" relationship. "Is a" relationships indicate that one entity extends another. For example, say there exists a person entity and an employee entity. Employees are all people (in most companies), thus they need the same attributes as people, so we will use a one-to-one relationship: employee is a person. It would be illogical (if not illegal with the labor authorities) to say that an employee is more than one person or that one person is two employees.

Many-to-Many Relationships

The final type of binary relationship is the many-to-many relationship. Instead of a single parent and one or more children, there would be more than one parent with more than one child. For example, a child has (biologically, at least) more than one parent: a mother and a father. This mother and father may have more than one child, and each mother and father can have children from other relationships as well.

Another common example of a many-to-many relationship is a car dealer. Pick nearly any single model of car, and you'll see that it is sold by many different car dealers. Similarly, one car dealer sells many different car models.

Note also that a large number of relationships between entities you will encounter will be many-to-many relationships. For example, it seems that relationships like an album's relationship to a song is simply one-to-many when you begin defining your entities, yet a song can be on many albums. Once you start to include concepts such as singers, musicians, writers, and so on, into the equation, you will see that it requires a lot of many-to-many relationships to adequately model those relationships (many singers to a song, many songs to a singer, etc.). An important part of the design phase is to examine the cardinality of your relationships and make sure you have considered how entities relate to one another in reality, as well as in your computer system.

The many-to-many relationship is not directly implementable using a simple SQL relationship but is typically implemented by introducing another table to implement the relationship. Instead of the key from one table being migrated to the other table, the keys from both tables in the relationship are migrated to a new table that is used to implement the relationship. In Chapter 3, I'll present more examples and discuss how to implement the many-to-many relationship.

Working with Nonbinary Relationships

Nonbinary relationships involve more than two tables in the relationship. Nonbinary relationships can be very problematic to discover and model properly, yet they are far more common than you might expect, for example:

> *A room is used for an activity in a given time period.*
> *Publishers sell books through bookstores and online retailers.*

Consider the first of these. We start by defining tables for each of the primary concepts mentioned—room, activity and time period:

```
Room (room_number)
Activity (activity_name)
Time_Period (time_period_name)
```

Next, each of these will be connected in one table to associate them all into one relationship:

```
Room_Activity_TimePeriod (room number, activity_name, time_period_name)
```


www.it-ebooks.info

We now have a table that represents the relationship of room, activity, and time utilization. From there, it may or may not be possible to break down the relationships between these three tables (commonly known as a ternary relationship, because of the three tables) further into a series of relationships between the entities that will satisfy the requirements in an easy-to-use manner. Often, what starts out as a complex ternary relationship is actually discovered to be a couple of binary relationships that are easy to work with. This is part of the normalization process that will be covered in Chapter 5. During the early, conceptual phases of design, it is enough to simply locate the existence of the different types of relationships.

Understanding Dependencies

Beyond basic database terms, I want to introduce a few mathematical concepts now before they become necessary later. They center on the concept of dependencies. The structure of a database is based on the idea that given one value, you can find related values. For a real-world example, take a person. If you can identify the person, you can also determine other information about the person (such as hair color, eye color, height, or weight). These values for each of these attributes may change over time, but when you ask the question, there will be one and only one answer to the question. For example, at any given instant, there can be only one answer to the question, "What color are the person's eyes?"

We'll discuss two different concepts related to this in the sections that follow: functional dependencies and determinants. Each of these is based on the idea that one value depends on the value of another.

Working with Functional Dependencies

Functional dependency is a very simple but important concept. It basically means that if you can determine the value of attribute A given a value of attribute B, B is functionally dependent on A. For example, say you have a function, and you execute it on one value (let's call it Value1), and the output of this function is always the same value (Value2). Then Value2 is functionally dependent on Value1. When a function always returns the same value for the same input, it is considered deterministic. On the other hand, if the value from the function can vary for each execution, it is nondeterministic.

In a table, consider the functional dependency of nonkey columns to key columns. For example, consider the following table T with a key of column X:

```
X   Y
--- ---
1   1
2   2
3   2
```

You can think of column Y as functionally dependent on the value in X, or fn(x) = y. Clearly, Y may be the same for different values of X, but not the other way around. This is a pretty simple yet important concept that needs to be understood.

```
X   Y   Z
--- --- ---
1   1   20
2   2   4
3   2   4
```


www.it-ebooks.info

In this example, fn(x) = y and fn(x) = z for certain, but looking at the data, there also appears to exist another dependency in this small subset of data, f(y) = z. Consider that f(y) = z, and you want to modify the z value to 5 for the second row:

```
X   Y   Z
--- --- ---
1   1   20
2   2   5
3   2   4
```

Now there is a problem with our stated dependency of f(y) = z because f(2) = 5 AND 4. As you will see quite clearly in Chapter 5, poorly understood functional dependencies are at the heart of many database problems, because one of the primary goals of any database design is that to make one change to a piece of data you should not need to modify data in more than one place. It is a fairly lofty goal, but ideally, it is achievable.

Working with Determinants

A term that is related to functional dependency is "determinant," which can be defined as "any attribute or set of attributes on which any other attribute or set of attributes is functionally dependent." In our previous example, X would be considered the determinant. Two examples of this come to mind:

- Consider a mathematical function like 2 * X. For every value of X, a particular value will be produced. For 2, you will get 4; for 4, you will get 8. Anytime you put the value of 2 in the function, you will always return a 4, so 2 functionally determines 4 for function (2 * X). In this case, 2 is the determinant.
- In a more database-oriented example, consider the serial number of a product. From the serial number, additional information can be derived, such as the model number and other specific, fixed characteristics of the product. In this case, the serial number functionally determines the specific, fixed characteristics, and as such, the serial number is the determinant in this case.

If this all seems familiar, it is because any key of a table will functionally determine the other attributes of the table, and each key will be a determinant, since it functionally determines the attributes of the table. If you have two keys, such as the primary key and alternate key of the table, each will be a determinant of the other.

Relational Programming

One of the more important aspects of relational theory is that there must be a high-level language through which data access takes place. As discussed earlier in this chapter, Codd's fifth rule states that "…there must be at least one language whose statements are expressible, by some well-defined syntax, as character strings, and whose ability to support all of the following is comprehensive: data definition, view definition, data manipulation (interactive and by program), integrity constraints, and transaction boundaries (begin, commit, and rollback)."

This language has been standardized over the years as the SQL we know (and love!). Throughout this book, we will use most of the capabilities of SQL in some way, shape, or form, because any discussion of database design and implementation is going to be centered around using SQL to do all of the things listed in the fifth rule and more.

www.it-ebooks.info

SQL is a relational language, in that you work on at the relation (or table) level on sets of data at a time, rather than on one row at a time. This is an important concept. Recall that Codd's seventh rule states "[t]he capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update, and deletion of data."

What is amazingly cool about SQL as a language is that one very simple statement almost always represents hundreds and thousands of lines of code being executed. Most of this code executes in the hardware realm, accessing data on disk drives, moving that data into registers, and performing operations in the CPU (envision Tim Allen on *Home Improvement* grunting here, please).

The tough part about SQL is that, as a relational language, it is very different than most other languages you may already know, like C# and Visual Basic. Using C#, you generally perform each action one at a time. You tell the computer to do one operation, and when that finishes, you do the next operation. The operations you do are usually quite low level and very specific. If you want to search for something in a file, you read the file, compare the value to the search criteria, and move to the next bit of data from the file.

SQL is a relational language and is a lot more restrictive in what you can do. You have two sorts of commands:

- *Data Definition Language (DDL)*: SQL statements used to set up data storage (tables), apply security, and so on.
- *Data Manipulation Language (DML)*: SQL statements used to create, retrieve, update and delete data that has been placed in the tables. In this book, I assume you have used SQL before, so you know that most everything done is handled by four statements: SELECT, INSERT, UPDATE, and DELETE.

As a relational programmer, your job is to give up control of all of the details of storing data, querying data, modifying existing data, and so on. The system (commonly referred to as the relational engine) does the work for you—well, a lot of the work for you. That statement "does the work for you" is very misleading and is a very common source of confusion. If you create your database correctly for the engine's needs, the engine will do the work for you, even for extremely large sets of data. And don't think the engine is simple, because it isn't. As Dr. David DeWitt (a technical fellow in the Data and Storage Platform Division at Microsoft Corporation) said, during his PASS Keynote in 2010, that query optimization (the process of taking a query and choosing the best way to execute it) isn't rocket science; it is far more difficult than that, mostly because people throw ugly queries at the engine and expect perfection.

The last point to make again ties back to Codd's rules, this time the twelfth, the nonsubversion rule. Basically, it states that the goal is to do everything in a language that can work with multiple rows at a time and that low-level languages shouldn't be able to bypass the integrity rules or constraints of the engine. In other words, leave the control to the engine and use SQL

Of course, this rule does not preclude other languages from existing. The twelfth rule does state that all languages that act on the data must follow the rules that are defined on the data. In some relational engines, it can be faster to work with rows individually rather than as sets. However, the creators of the SQL Server engine have chosen to optimize for set-based operations. This leaves the onus on the nonrelational programmer to play nice with the relational engine and let it do a lot of the work.

Outlining the Database-Specific Project Phases

As we go though the phases of a project, the database parts have some very specific names that have evolved to describe the models that are created. Much like the phases of the entire project, the phases that you will go through when designing a database are defined specifically to help you think about only what is necessary to accomplish the task at hand.

Good design and implementation practices are essential for getting to the right final result. Early in the process, your goals should simply be to figure out the basics of what the requirements are asking of you.

www.it-ebooks.info

Next, think about how to implement using proper fundamental techniques, and finally, tune the implementation to work in the real world. One of the biggest problems that relational database technology has in the eyes of many programmers is that it has been around for a very long time, over 40 years. In dog years that is 280 years, and in computer science terms, that is eternity (remember your calculator back then? Slide rule? Abacus?). Back when I started programming 18 years ago, things were much different because the technology was a lot weaker, so we all learned a lot of awesome tips and tricks to make it work. Well, here in the 2011, the technology is far better, so many of those tips and tricks aren't necessary anymore. Of course, we do work with much larger amounts of data than we did back then too (my first enterprise size database was on a server with 100MB (yes, 100 megabytes)). SQL itself still looks very much like it did back then, but the engine underneath is far more advanced. Not only are the old tricks obsolete but lots of new, very reasonable techniques can now be used with storage and processors to help increase performance without stepping back from sound design techniques.

If it seems like I am being evasive about the tricks of the old days, I am. If you know what they are, fine. If not, I don't want to start out this book telling you the wrong way to do things. The process I outline here steers us through creating a database by keeping the process focused on getting things done right:

- *Conceptual*: During this phase, the goal is a sketch of the database that you will get from initial requirements gathering and customer information. During this phase, you identify what the user wants. You work to find out as much as possible about the business process for which you are building this data model, its scope, and, most important, the business rules that will govern the use of the data. You then capture this information in a conceptual data model consisting of at least the set of high-level entities and the interactions between them.
- *Logical*: The logical phase is an implementation-nonspecific refinement of the work done in the conceptual phase, transforming what is often a loosely structured conceptual design into a full-fledged relational database design that will be the foundation for the implementation design. During this stage, you flesh out the model of the data that the system needs and capture all of the data business rules that will need to be implemented.
- *Physical*: In this phase, you adapt the logical model for implementation to the host RDBMS, in our case, SQL Server.
- *Storage*: In this phase, you create the model where the implementation data structures are mapped to storage devices. This phase is also more or less the performance tuning/optimization phase of the project, because it is important that your implementation should function (in all ways except performance) the same way no matter what the hardware looks like. It might not function very fast, but it will function. During this phase of the project—and not before this—indexes, disk layouts, and so on come into play .

Conceptual Phase

The conceptual design phase is essentially a process of analysis and discovery; the goal is to define the organizational and user data requirements of the system. Note that parts of the overall design picture beyond the needs of the database design will be part of the conceptual design phase (and all follow-on phases), but for this book, the design process will be discussed in a manner that may make it sound as if the database is all that matters (as a reader of this book who is actually reading this chapter on fundamentals, you probably feel that way already).

When writing this edition of this book, I discovered that the term "conceptual model" has several meanings depending on the person I interviewed about the subject. In some people's eyes, the conceptual model was no more than entities and relationships. Others included attributes and keys. My particular process has been to produce a conceptual model that simply includes the data that I locate in documentation. To stop at an arbitrary point in the process is not natural for most people. In the end, a conceptual entity is *not* the same

www.it-ebooks.info

thing as a table, nor can the attributes on a conceptual model be compared to the columns on a table. A table is an implementation-specific SQL construct that follows a pattern established before the final logical model is produced. Sometimes, an entity will map directly to a table in the implementation, but often, it won't. Some conceptual entities will be too abstract to ever be implemented, and sometimes, they will map to two or more tables. It is a major mistake (if somewhat unavoidable because of human nature) at this point of the process to begin thinking about how the final database will look.

The core activity that defines the conceptual modeling process for every source I have found is discovering and documenting a set of entities and the relationships among them. The goal being to capture, at a high level, the fundamental types of data that are required to support the business processes and users' needs. Entity discovery is at the heart of this process. Entities correspond to nouns (people, places, and things) that are fundamental to the business processes you are trying to improve by creating software.

Beyond this, how much more you model or document is very much debatable. The people that I have worked with have almost always documented as much of the information that they can in the model as they find it. My primary requirement for this part of the modeling phase is that you remember the name is conceptual and resist the need to start implementing the final database immediately.

Logical Phase

The logical phase is a refinement of the work done in the conceptual phase. The output from this phase will be an essentially complete blueprint for the design of the relational database. Note that during this stage, you should still think in terms of entities and their attributes, rather than tables and columns, though in the database's final state there may be basically be no difference. No consideration should be given at this stage to the exact details of how the system will be implemented. As previously stated, a good logical design could be built on any RDBMS. Core activities during this stage include the following:

- Drilling down into the conceptual model to identify the full set of entities that will be required to define the entire data needs of the user.
- Defining the attribute set for each entity. For example, an Order entity may have attributes such as Order Date, Order Amount, Customer Name, and so on.
- Applying normalization rules (covered in Chapter 5).
- Identifying the attributes (or a group of attributes) that make up candidate keys (i.e., sets of attributes that could uniquely identify an instance of an entity). This includes primary keys, foreign keys, surrogate keys, and so on.
- Defining relationships and associated cardinalities.
- Identifying an appropriate domain (which will become a datatype) for each attribute and whether values are required.

While the conceptual model was meant to give the involved parties a communication tool to discuss the data requirements and to start seeing a pattern to the eventual solution, the logical phase is about applying proper design techniques. The logical modeling phase defines a blueprint for the database system, which can be handed off to someone else with little knowledge of the system to implement using a given technology (which in our case is going to be some version of Microsoft SQL Server, most likely 2011).

■ **Note** Before we begin to build the logical model, we need to introduce a complete data modeling language. In our case, we will be using the IDEF1X modeling methodology, described in Chapter 3.

---

www.it-ebooks.info

Physical

During the physical implementation phase, you fit the logical design to the tool that is being used (in our case, the SQL Server RDBMS). This involves choosing datatypes, building tables, applying constraints and writing triggers to implement business rules, and so on to implement the logical model in the most efficient manner. This is where platform-specific knowledge of SQL Server, T-SQL, and other technologies becomes essential.

Occasionally, this phase can entail some reorganization of the designed objects to make them easier to implement in the RDBMS. In general, I can state that for most designs there is seldom any reason to stray a great distance from the logical model, though the need to balance user load and hardware considerations can make for some changes to initial design decisions. Ultimately, one of the primary goals is that no data that has been specified or integrity constraints that have been identified in the conceptual and logical phases will be lost. Data can (and will) be added, often to handle the process of writing programs to use the data. The key is to avoid affecting the designed meaning or, at least, not to take anything away from that original set of requirements.

At this point in the project, constructs will be applied to handle the business rules that were identified during the conceptual part of the design. These constructs will vary from the favored declarative constraints, such as defaults and check constraints, to less favorable but still useful triggers and occasionally stored procedures. Finally, this phase includes designing the security for the data we will be storing.

Storage Phase

The goal of the storage layout phase is to optimize data access—for example, by implementing effective data distribution on the physical disk storage and by judicious use of indexes. While the purpose of the RDBMS is to largely isolate us from the physical aspects of data retrieval and storage, in this day and age it is still very important to understand how SQL Server physically implements the data storage to optimize database access code.

During this stage, the goal is to optimize performance *without* changing the implemented database in any way to achieve that aim. This goal embodies Codd's eleventh rule, which states that an RDBMS should have distribution independence. Distribution independence implies that users should not have to be aware of whether a database is distributed. Distributing data across different files, or even different servers, may be necessary, but as long as the published physical object names do not change, users will still access the data as columns in rows in tables in a database.

■ **Note** Our discussion of the storage model will be reasonably limited. We will start by looking at entities and attributes during conceptual and logical modeling. In implementation modeling, we will switch gears to deal with tables, rows, and columns. The physical modeling of records and fields will be dealt with only briefly (in Chapter 8). If you want a deeper understanding of the physical implementation, check out "Microsoft® SQL Server® 2008 Internals" by Kalen Delaney (Microsoft Press, 2006) or any future books she may have released by the time you are reading this. If you want to know where the industry is headed, take a look at the SQL Azure implementation, where the storage aspects of the implementation are very much removed from your control and even grasp.

Summary

In this chapter, I offered a quick history to provide context to the trip I will take you on in this book, along with some information on the basic concepts of database objects and some aspects of theory. It's very important that you understand most of the concepts discussed in this chapter, since from now on, I'll assume you understand them, though to be honest, all of the really deeply important points seem to come up over and over and over

www.it-ebooks.info

throughout this book. I pretty much guarantee that the need for a natural key on every table will be repeated enough that you may find yourself taking vacations to the Florida Keys and not even realize why (hint, it could be the *natural key* beauty that does it do you).

Next, we went over a road map for the different phases our database design. This road map will, in fact, be the process that will be used throughout the rest of this book and is how the book is organized. The road map phases are as follows:

- *Conceptual*: Identify at a high level what the users need in the database system you are creating.
- *Logical*: Design the database, and prepare it for implementation.
- *Physical:* Design and implement the database in terms of the tools used (in the case of this book, SQL Server 2012), making adjustments based on the realities of the current version of SQL Server/other RDBMS you are working with.
- *Storage*: Design and lay out the data on storage based on usage patterns and what works best for SQL Server. The changes made to the storage layer ought to only affect performance, not correctness.

Of course, the same person will not necessarily do every one of these steps. Some of these steps require different skill sets, and not everyone can know everything—or so we're told.

I introduced relational data structures and defined what a database is. Then, we covered tables, rows, and columns. From there, I explained the information principle (which states that data is accessible *only* in tables and that tables have no order), defined keys, and introduced NULLs and relationships. We also looked at a basic introduction to the impetus for how SQL works.

Finally, we discussed the concept of dependencies, which basically are concerned with noticing when the existence of a certain value requires the existence of another value. This information will be used again in Chapter 3 as we reorganize our data design for optimal usage in our relational engine.

In the next few chapters, as we start to formulate a conceptual and then a logical design, we will primarily refer to entities and their attributes. After we have logically designed our tables, we'll shift gears to the implementation phase and speak of tables, rows, and columns. Here is where the really exciting part comes, because the database construction starts, and our database starts to become real. After that, all that is left is to load our data into a well-formed, well-protected relational database system and set our users loose!

Everything starts with the fundamentals presented here, including understanding what a table is and what a row is (and why it differs from a record. As a last not-so-subtle-subliminal reminder, rows in tables have no order, and tables need natural keys).

www.it-ebooks.info

CHAPTER 2

Introduction to Requirements

Whatever pursuit you undertake, the requirements should start with a love of what it is that you are pursuing.

—Bill Toomey

When tasked to design a database system, you have to be mindful that users often aren't technologists. Sometimes, you may even have to understand that people with organizational power or check signing rights may not have the proper amount of intelligence doled out to them. So when you start to gather requirements, relying on the client's management professionals to know what they want and how to implement it is almost always complete insanity. On the flip side, keep in mind that you're building a system to solve a business problem first for the users, then for the people who sign the checks, and not for yourself. What the users (and check writers) want is the most important, so you must consider that first and foremost.

There's an old saying that you shouldn't build users a Cadillac when all they want is a Volkswagen (though when this saying was coined, a VW wasn't quite as expensive as it is today). But overdelivering is a lot better than underdelivering. Consider how excited your client would be if instead of the Volkswagen you had given them a 1983 Yugo (now or then, it doesn't truly matter!). While the concepts behind these vehicles are similar (four wheels, steering wheel, etc.), erring on the side of quality is usually better, if err you must. Admittedly more often the problem is that the clients wanted a car and what gets delivered is something more akin to a bowl of fruit. Even the nicest bowl of fruit isn't going to please your users if they paid thirty large for a mode of transportation.

The next problem is that users don't think about databases; they think about user interfaces (UIs) and reports. Of course, a lot of what the user specifies for a UI or report format is actually going to end up reflected in the database design; it is up to you to be certain that there is enough in the requirements to design storage without too much consideration about how it will be displayed, processed, or used. The data has an essence of its own that must be obeyed at this point in the process, or you will find yourself in a battle with the structures you concoct. In this chapter, we will go through some of the basic sorts of data you want to get and locations to look to make sure you are gathering the right kinds of requirements to begin the database design process.

Of course, if you are a newbie, you are probably thinking that this all sounds like a lot of writing and not a lot of coding. If so, you get a gold star for reading comprehension. No matter how you slice it, planning the project is like this. If you are lucky, you will have analysts who do the requirements gathering so you can design and code software. However, the importance of making sure someone gathers requirements cannot be understated.

www.it-ebooks.info

During a software project (and really any project, but let's focus on software projects), there are phases that are commonly gone through:

- *Requirement gathering*: Document what a system is to be, and identify the criteria that will make the project a success.
- *Design*: Translate the requirements into a plan for implementation.
- *Implementation*: Code the software.
- *Testing*: Verify that the software does what it is supposed to do.
- *Maintenance*: Make changes to address problems not caught in testing.

Each phase of the project after requirement gathering relies on the requirements to make sure that the target is met. Requirements are like a roadmap, giving you the direction and target to get there. Trying to build your database without first outlining those requirements is like taking a trip without your map. The journey may be fun, but you may find you should have taken that left turn at Albuquerque, so instead of sunning your feathers on Pismo beach, you have to fight an abominable snowman. Without decent requirements, a very large percentage of projects fail to meet the users' needs. A very reasonable discussion that needs to be considered is how many requirements are enough. In the early days of software development, these phases were done one at a time for the entire project, so you gathered all requirements that would ever be needed and then designed the entire software project before any coding started, and so on. This method of arranging a project has been given the somewhat derogatory name of "waterfall method" because the output of one step flowed into another.

The important point I want to make clear in this chapter is simple. The waterfall process in total may have been a complete failure, but the steps involved were not. Each of these steps will be performed whether you like it or not. I have been on projects where we started implementation almost simultaneously with the start of the project. Eventually, we had to go back to gather requirements to find out why the user wasn't happy with our output. And the times when we jumped directly from gathering requirements to implementation were a huge mess, because every programmer did his or her own thing, and eventually every database, every object, and every interface in the system looked completely different. It is a mess that is probably still being dug out from today.

This book is truly about design and implementation, and after this chapter, I am going to assume requirements are finished, and the design phase has begun. Many books have been written about the software requirements gathering and documenting process, so I am not going to even attempt to come up with an example of requirements. Rather, I'll just make a quick list of what I look for in requirements. As writer Gelett Burress once said about art, "I don't know anything about art, but I know what I like," and the same is really quite true when it comes to requirements. In any case, requirements should be captured, and you can generally tell the good from the bad by a few key criteria:

- Requirements should generally include very few technical details about how a problem will be solved; they should contain only the definition of the problem and success criteria. For example, a good requirements document might say "the clerks have to do all of their adding in their heads, and this is slow and error prone. For project success, we would prefer the math done in a manner that avoids error." A poor requirements document would exchange the last phrase for ". . . we would prefer the math be done using a calculator." A calculator might be the solution, but the decision should be left to the technologist.
- The language used should be a specific as possible. As an example, consider a statement like "we only pay new-hire DBAs $20,000 a year, and the first raise is after six months." If this was the actual requirement, the company could never hire a qualified DBA—ever. And if you implemented this requirement in the software as is, the first time the company wanted to break the rule (like if Paul Nielsen became available), that user would curse your name, hire Paul as a CEO, and after six months, change his designation to DBA.

www.it-ebooks.info

(Users will find a way.) If the requirement was written specifically enough, it would have said "We usually only. . .", which is implemented much differently.

- Requirements should be easily read and validated by customers. Pure and simple, use language the users can understand, not technical jargon that they just gloss over so they don't realize that you were wrong until their software fails to meet their needs.

For my mind, it really doesn't matter how you document requirements, just as long as they get written down. Write them down. Write them down. Hopefully, if you forget the rest of what I said in this chapter, you'll remember that. If you are married or have parents, you have probably made the mistake of saying, "Yes _____, I promise I will get that done for you" and then promptly forgetting what was said exactly so an argument eventually occurs. "Yes, you did say that you wanted the screen blue!" you say to your customers. At this point, you have just called them liars or stupid, and that is not a great business practice. On the other hand, if you forward the document in which they agreed to color the screen blue, taking responsibility for their mistake is in their court.

Finally, how will we use written requirements in the rest of the software creation process? In the design phase, requirements are your guide to how to mold your software. The technical bits are yours (or corporate standards) to determine: two tables or three, stored procedures or ad hoc access, C# or VB? But the final output should be verifiable by comparing the design to the requirements. And when it is time to do the overall system tests, you will use the requirements as the target for success.

In this chapter, I will cover two particular parts of the requirements gathering process:

- *Documenting requirements*: I'll briefly introduce the types of concerns you'll have throughout the project process in terms of documenting requirements.
- *Looking for requirements*: Here, I'll talk about the places to find information and some techniques for mining that information.

Requirements are not a trivial part of a project, and most certainly should not be omitted, but like anything, they can be overdone. This chapter will give you a bit of advice on where to look, or if you are in the happy programmer position of not being the one gathering requirements, what to make sure has been looked at. The sad reality of programming is that the system you create stinks because the requirements that you are given stink, it won't be the requirements gatherer who gets to recode.

Documenting Requirements

If you've ever traveled to a place where no one speaks the same language as you, you know the feeling of being isolated based solely on communication. Everything everyone says sounds weird to you, and no matter how often you ask where the bathroom is, all you get is this blank look back. It has nothing to do with intelligence; it's because you aren't speaking the same language. A word of advice: you can't expect the entire population of another country to learn your language perfectly just so you can get what you need. It works better if you learn their language. Even when two people speak the same basic language, often there can be dialects and phrasing that can be confusing.

Information technology professionals and our clients tend to have these sorts of communication issues, because frequently, we technology types don't speak the same dialect or even the same language as our clients. Clients tend to think in the language of their industry, and we tend to think in terms of computer solutions. You probably have the same feelings when you are the user as they do. For example, think about SQL Server's tools. We relational programmers have trouble communicating to the tool designers what we want in SQL Server's tools. They do an adequate job for most tasks, but clearly, they aren't completely on the same page as the users.

During the process of analysis, you should adopt one habit early on: document, document, document as much of the information that you acquire as reasonably possible. It's horrible to think about, but your coworkers

www.it-ebooks.info

might get hit by a bus tomorrow, and every bit of information in their heads will be rendered useless while they recover. Less morbidly (I guess, depending on how you feel about your coworkers), if a project team member decides to take a month off, someone else will have to take over his or her work (or you just might have to wait a month to make any progress, leading to long, working weekends). So, you should document, document, document; do it during a meeting and/or immediately after it. Without documentation, you will quickly risk forgetting vital details. It's imperative that you don't try to keep everything in your head, because even people with the best memories tend to forget the details of a project (especially if they're hit by that bus I talked about earlier).

The following are a few helpful tips as you begin to take notes on users' needs:

- Try to maintain a set of documents that will share system requirement and specification information. Important documents to consider include design-meeting notes, documents describing verbal change requests, and sign-offs on all specifications, such as functional, technical, testing, and so on.
- Beyond formal documentation, it's important to keep the members of your design team up to date and fully informed. Develop and maintain a common repository for all the information, and keep it up to date.
- Note anywhere that you add information that the users haven't given you or outwardly agreed to.
- Set the project's scope early on, and keep it in mind at all times. This will prevent the project from getting too big or diverse to be achievable within a reasonable period of time and within the budget. Hashing out changes that affect the budget, particularly when it will increase the budget, early in the process will avoid future animosity.

Once you document something, a crucial step follows: make sure the client agrees with your version of the documentation. As you go through the entire system design process, the clients will no doubt change their minds on entities, data points, business rules, user interface, colors—just about anything they can—and you have to prepare yourself for this. Whatever the client wants or needs is what you have to endeavor to accomplish. The client is in ultimate control of the project, which unfortunately often means communicating through a third party like a project manager and being flexible enough to run with any proposed changes, whether minor or major. What can be even worse is that you may not get to deal with the client directly at all, but only through a project manager. This setup initially sounds great, because you think the project manager will translate for you and be on the side of quality and correctness, and sometimes this is true. But often, the manager will mistranslate a client desire into something quite odd and then insist that it is the client's desire. "I need all of the data on one screen" gets translated into "I need all of the data in one table." Best case is that the manager realizes who the technical people are and who have business needs. If you have a typical job, worst case is what you experience every day.

Of course, it is a reality that clients change their minds, and sometimes, it seems to be more than a daily occurrence. Most frequently, they want more and more features. The common term for this is "scope creep." The best way to avoid conflict is to make sure you get your client's approval at regular stages throughout the design process. This is sometimes known as the principle of CYA, which I think has something to do with covering all your bases, though the letters may have a more interesting meaning. If you have no written history, telling your client that they are wrong is translated to "liar!" With documentation it translates to a far nicer "it is OK, I forget stuff too." Users are humans too!

In addition to talking to the client, it's important to acquire as many notes, printouts, screen shots, CD-ROMs loaded with spreadsheets, database backups, Word documents, e-mails, handwritten notes, and so on that exist for any current solution to the problem. This data will be useful in the process of discovering data elements, screens, reports, and other elements that you'll need to design into your applications. Often, you'll find information in the client's artifacts that's invaluable when putting together the data model.

www.it-ebooks.info

■ **Tip** Throughout the process of design and implementation, you'll no doubt find changes to the original requirements. Make sure to continue to update your documentation, because the most wonderfully written and formatted documentation in the world is useless if it's out of date.

Gathering Requirements

Without the designer having a solid understanding of the requirements, the system will essentially be based on guesses. Imagine your parent or spouse asking you to paint a room. Would that be enough to get started? What color paint? What type of paint? (Did you know you can buy paint that turns a wall into a white board?) There are many, many questions you would immediately ask for such a simple problem. If you ever pay people to paint your house, the painters will go over every detail of the process with you, from paint to trim, making sure they know what you desire. Painting a house takes a lot of skill, but the overall process is so much less complex than even the most simple computer system. However, a lot of people take about the same amount of time gathering requirements for a complex program (and it takes a day or two to paint a house).

It isn't necessary to gather every requirement about every area of a large system initially; the system can be broken down into portions, often referred to as subject areas. The size of the subject area is based on the needs of the team and development methodology used. For example, the Scrum methodology breaks down everything into (generally speaking) 30-day units for designing, coding, and testing, while something like the waterfall methodology would expect you to design the entire system first and then start coding. If Scrum were your team's methodology, the subject area might be small, function-oriented subject areas based on a set of user needs. Or you might break things down into larger functional areas, such as an accounts-payable module in an accounting system or a user-management module for a web site. The important thing is that all development methodologies will tell you one thing: design *before* you code. Sure, there is a good amount of variability when it comes to how much design you need, but you still don't start typing until you know where you are going.

For gathering requirements, there are many tools and methodologies for documenting processes, business rules, and database structures. The Unified Modeling Language (UML) is one possible choice and there are also several model types in the IDEF family of methods; we will cover the data modeling technique in Chapter 3. I'll employ the Entity-Relationship (E-R) modeling method IDEF1X to model databases. I won't be covering any of the other modeling languages for the nondatabase structure parts of the project but will rather be using a simple manual spreadsheet method, which is by far the most common method of documenting requirements—even in medium-sized organizations where spending money on documentation tools can be harder than teaching your cat to make good word choices when playing Scrabble.

Regardless of the tools used to document the requirements, the needs for the database design process are the same. Specifications need to be acquired for the following:

- Entities and relationships
- Attributes and domains
- Business rules that can be enforced in the database
- Processes that require the use of the database

Without these specifications, you'll either have to constantly go back to the clients and ask a bunch of questions (which they will sometimes answer three different ways for every two times they are asked, teaching you discernment skills) or start making guesses. Although guessing wrong a few times is a good education in how not to do things, it's certainly no way to work efficiently (unless you happen to be the Amazing Kreskin and guess right 99.9% of the time, though I am pretty sure it was a trick and he had done his requirements gathering as well).

www.it-ebooks.info

As a major part of the process of implementing a database system, the data architect's goal will be to produce a graphical model of the database. (As stated in the previous chapter, I'll be using IDEF1X–based data model diagrams, but you can use whatever methodology your tool supports, even if it is just a piece of paper and a No. 2 pencil.)

■ **Tip** During the early parts of a project, figure out the "what" and "why" first; then you can work on the "how." Once you know the details of what needs to be built, the process to get it built will be reasonably natural, and you can possibly apply preexisting patterns to the solution.

Vagueness may cause unnecessary discussions, fights, or even lawsuits later in the process. So, make sure your clients understand what you're going to do for them, and use language that will be clearly understood but that's specific enough to describe what you learn in the information gathering process.

Throughout the process of discovery, *artifacts* will be gathered and produced that will be used throughout the process of implementation as reference materials. Artifacts are any kind of documents that will be important to the design, for example, interview notes, e-mails, sample documents, and so on. In this section, I'll discuss the some of the main types of artifacts or activities that you will need to be very interested in as a database architect:

- Client interviews
- Prototypes and existing systems
- Various other types of documentation

By no means is this an exhaustive list of where to find and acquire documentation; in fact, it's far from it. The goal is simply to get your mind clicking and thinking of information to get from the client so your job will be easier.

Interviewing Clients

It might be the case that the person designing the data storage (commonly referred as the data architect) will never meet the user, let alone be involved in formal interviews. The project manager, business analyst, and/or system architect might provide all the required information. Other projects might involve only a data architect or a single person wearing more hats than the entire Fourth Army on maneuvers. I've done it both ways: I've been in the early design sessions, and I've worked from documentation. The better the people you work with, the more favorable the latter option is. In this section, I'll talk quickly about the basics of client interviews, because on almost any project, you'll end up doing some amount of interviewing the client.

Client interviews are commonly where the project really gets started. It's where the free, unstructured flow of information starts. However, it's also where the communication gap starts. Many clients generally think visually—in terms of forms, web pages, and perhaps in simple user interfaces. Users also tend to think solely from their own perspective. For example, they may use the word "error" to denote why a process did not run as they expected. These error conditions may not only be actual errors but choices the user makes. So a value like "scheduled maintenance" might be classified as an error condition. It is very much up to the people with "analyst" embroidered on the back of their hats to analyze what users are actually asking for.

As such, the job is to balance the customers' perceived wants and needs with their real need: a properly structured database that sits nicely behind a user interface and captures what they are really after, specifically information to make their business lives easier and more lucrative. Changing a form around to include a new text box, label, or whatever is a relatively simple task, giving the user the false impression that creating the entire application is an easy process. If you want proof, make the foolish mistake of demonstrating a polished-looking

www.it-ebooks.info

prototype application with non-hard-coded values that makes the client think it actually works. The clients might be impressed that you've put together something so quickly and expect you to be nearly done. Rarely will they understand that what exists under the hood—namely, the database and the middle-tier business objects—is where all the main work takes place.

■ **Tip** While visual elements are great places to find a clue to what data a user will want, as you go along in the process, you'll want to be careful not to center your database design too heavily around a particular interface. The structure of the data needs to be dictated by what the data means, not on how it will be presented. Presentation is more of an interface design task, not a database design one.

Brainstorming sessions with users can also yield great results for gathering a lot of information at once, as long as the group doesn't grow too large (if your meeting requires an onsite caterer for lunch, you are not going to make any great decisions). The key here is to make sure that someone is facilitating the meeting and preventing the "alpha" person from beating up on the others and giving only his or her own opinion (it is even worse if you are that alpha person!). Treat information from every person interviewed as important, because each person will likely have a different, yet valuable viewpoint. Sometimes (OK, usually) the best information comes not from the executive, but from the person who does the work. Don't assume that the first person speaks for the rest, even if they're all working on the same project or if this individual is the manager (or even president or owner of a major corporation, though a great amount of tact is required sometimes to walk that tightrope).

In many cases, when the dominant person cannot be controlled or the mousey person cannot be prodded into getting involved, one-on-one sessions should probably be employed to allow all clients to speak their minds, without untimely interruptions from stronger-willed (though sometimes not stronger-minded) colleagues. Be mindful of the fact that the loudest and boldest people might not have the best ideas and that the quiet person who sits at the back and says nothing might have the key to the entire project. Make sure to at least consider everybody's opinions.

This part of the book is written with the most humility, because I've made more mistakes in this part of the design process than any other. The client interview is one of the most difficult parts of the process that I've encountered. It might not seem a suitable topic for experienced analysts, but even the best of us need to be reminded that jumping the gun, bullying the clients, telling them what they want before they tell you, and even failing to manage the user's expectations can lead to the ruin of even a well-developed system. If you have a shaky foundation, the final product will likely be shaky as well.

Asking the Right Questions

When painting a house, there are a set of questions that the painting company's representative will ask every single one of their clients (colors to use? rooms to paint?). The same can go for almost any computer software project. In the following sections are some questions that are going to be important to the database design aspects of a system's development. Clearly, this is not going to be an exhaustive list, but it's certainly enough to get you started, so at a minimum, you won't have to sit in a room one day with no idea about what to say.

What Data Is Needed?

If the data architect is part of the design session, some data is clearly needed for the system. Most users, at a high level, know what data they want to see out of the system. For example, if they're in accounting, they want to see dollars and cents summarized by such-and-such a group. It will be very important at some time in your

www.it-ebooks.info

process to differentiate between what data is needed and what would just be nice to have. Whenever you start to do your design/implementation project plans (an activity that is 30% less fun than going to the dentist for a cleaning, and 10% more pleasant than a root canal), you can focus on the necessities and keep the wants in your pocket for later.

How Will the Data Be Used?

Knowing what your client is planning to use the data in the system for is an important piece of information indeed. Not only will you understand the processes that you will be trying to model, but you can also begin to get a good picture of the type of data that needs to be stored.

For example, imagine you're asked to create a database of contacts for a dental office. You might want to know the following:

- Will the contact names be used just to make phone calls, like a quick phone book?
- Will the client be sending e-mail or posting to the members of the contact lists? Should the names be subdivided into groups for this purpose?
- Will the client be using the names to solicit a response from the mail, such as appointment reminders?
- Is it important to have family members documented? Do they want to send cards to the person on important dates?

Usage probably seems like it would be out of bounds early in the design process, and in some ways, you would be right. But in broad strokes, usage information is definitely useful. Information about the types of processes where data might be used is important, but what screen it might show up on is less so. For example, take addresses. If you just capture them for infrequent usage, you might only need to give the user a single string to input an entire address. But if your business is mailing, you may need to format it to your post office's exact specifications, so you don't have to pay the same postage rates as the normal human beings.

What Rules Govern the Use of the Data?

Almost every piece of data you are going to want to store will have rules that govern how it is stored, used, and accessed. These rules will provide a lot of guidance to the model that you will produce. As an example, taking our previous example of contacts, you might discover the following:

- Every contact must have a valid e-mail address.
- Every contact must have a valid street address.
- The client checks every e-mail address using a mail routine, and the contact isn't a valid contact until this routine has been successfully executed.
- Contacts must be subdivided by the type of issues they have.
- Only certain users can access the e-mail addresses of the contacts.

It's important to be careful with the verbiage of the rules gathered early in the process. Many times, the kinds of rules you get seem pretty straightforward when they are written down, but the reality is quite often not so simple. It is really important as you are reviewing rules to confirm them with the analyst and likely directly with the client before assuming them to be true.

www.it-ebooks.info

As a case in point, what is a "valid" e-mail address? Well, it's the e-mail address that accurately goes with the contact. Sure, but how on Earth do you validate that? The fact is that in many systems you don't. Usually, this is implemented to mean that the string meets the formatting for an e-mail address, in that it has an ampersand character between other characters and a dot (.) between one or more alphanumeric values (such as %@%.%, plus all characters between A and Z, 0 and 9, an underscore, and so on), but the value is completely up to interpretation. On the other hand, in other types of systems, you actually require the user to pick up some information from the e-mail to validate that it is, indeed, a working e-mail address and that the person who entered the data has rights to it. It is very much up to the needs of the system, but the English question can easily be put using the exact same words.

The real problem comes when you too-strictly interpret rules and your final product ends up unacceptable because you've placed an overly restrictive rule on the data that the client doesn't want or you've missed a rule that the client truly needs. I made this mistake in a big way once, which torpedoed a system for several weeks early in its life. Rules that the clients had seemingly wanted to be strictly enforced needed to be occasionally overridden on a case-by-case basis, based on their clients' desires. Unfortunately, our program didn't make it possible for the user to override these rules, so teeth were gnashed and sleep was lost fixing the problem (even worse, it was a fixed-bid contract where these kinds of overages meant no more money, completely eating away any possible bonus. And no, I didn't make that deal).

Some rules might have another problem: the client wants the rule, but implementing it isn't possible or practical. For example, the client might request that all registered visitors of a web site have to insert a valid mobile phone number, but is it certain that visitors would provide this data? And what exactly is a valid mobile number? Can you validate that by format alone, or does the number have to be validated by calling it or checking with the phone company? What if users provide a landline instead? Implementability is of no concern at this point in the process. Someone will have to enforce the rule, and that will be ironed out later in the process.

What Data Is Reported On?

Reports are often one of the most frequently forgotten parts of the design process, yet in reality, they are almost certainly the most important part of the project to the client. Usually, the thing that makes a system profitable is the ability to report on all of the activity in the system. How productive different parts of the organization are, how effective sales persons are, and so on are a large part of the "why" that make computer systems worthwhile for even very small companies.

Many novice developers leave designing and implementing reports until the last minute (a mistake I've made more than once over the years). For the user, reports are where data becomes information and are used as the basis of vital decision making and can make or break a company.

Looking back at the contact example, what name does the client want to see on the reports? The following items come to mind:

- First name, last name
- First name, middle name, last name
- Last name, first name
- Nickname

It's important to try to nail down such issues early, no matter how small or silly they seem to you at this point. They're important to the client, who you should always remember is paying the bill. And frankly, the most important rule for reporting is that you cannot report on data that you do not capture.

From a database design standpoint, the content of reports is extremely important, because it will likely help to discover data requirements that aren't otherwise thought of. Avoid being concerned with the ascetics of the reports yet, because that might lead to the temptation of coding and away from modeling.

www.it-ebooks.info

■ **Tip** Don't overlook any existing reports that might have a corresponding report in the new system. Rarely should you just duplicate old reports, but the content will likely include data that the client may never even think about when they're expressing needs. There will often be hundreds of reports currently in production, and in the new system, there is little doubt that the number will go up, unless many of the reports can be consolidated.

Where Is the Data Now?

It is nice once in a while to have the opportunity to create a totally new database with absolutely no preexisting data. This makes life so easy and your job a lot of fun. Unfortunately, as years pass, finding a completely new system to implement gets less likely than the Cubs winning the World Series (no offense to Cubs fans, but hey, it is what it is). The only likely exception is when building a product to be sold to end users in a turnkey fashion (then the preexisting data is their problem, or yours if you purchase their system). For almost every system I have worked on, I was creating a better version of some other system, so we had to consider converting existing data that's important to the end users. (Only one major system was a brand new system. That was a wonderful experience for many reasons; not only didn't we have to deal with data conversion but we didn't have to deal with existing processes and code either.)

Every organization is different. Some have data in one centralized location, while others have it scattered in many (many) locations. Rarely, if ever, is the data already in one well-structured database that you can easily access. If that were the case, why would the client come to you at all? Clients typically have data in the following sundry locations:

- *Mainframe or legacy servers*: Millions of lines of active COBOL still run many corporations.
- *Spreadsheets*: Spreadsheets are wonderful tools to view, slice, and dice data but are wildly inappropriate places to maintain complex data. Most users know how to use a spreadsheet as a database but, unfortunately, are not so experienced in ensuring the integrity of their data, so this data is undoubtedly going to give you a major headache.
- *Desktop databases such as Microsoft Access*: Desktop databases are great tools and are easy to deploy and use. However, this ease of use often means that these databases are constructed and maintained by nontechnical personnel and are poorly designed, potentially causing many problems when the databases have to be enlarged or modified.
- *Filing cabinets*: Even now, in the twenty-first century, many companies still have few or no computers used for anything other than playing solitaire and instead maintain stockpiles of paper documents. Your project might simply be to replace a filing cabinet with a computer-based system or to supply a simple database that logs the physical locations of the existing paper documents.

Data that you need to include in the SQL Server database you're designing will come from these and other weird and wonderful sources that you discover from the client (truth is commonly stranger than fiction). Even worse, spreadsheets, filing cabinets, and poorly designed computerized databases don't enforce data integrity (and often desktop databases, mainframe applications, and even existing SQL Server databases don't necessarily do such a perfect job either), so always be prepared for dirty data that will have to be cleaned up before storage in your nice new database.

www.it-ebooks.info

Will the Data Need to Be Integrated with Other Systems?

Once you have a good idea of where the client's important data is located, you can begin to determine how the data in your new SQL Server solution will interact with the data that will stay in its original format. This might include building intricate gateway connections to mainframes, linking server connections to other SQL Servers or Oracle boxes, or even linking to spreadsheets. You can't make too many assumptions about this topic at this point in your design. Just knowing the architecture you'll need to deal with can be helpful later in the process.

■ **Tip** *Never* expect that the data you will be converting or integrating with is going to have *any* quality. Too many projects get their start with poor guesses about the effort required, and data cleanup has been the least well-guessed part of them all. It will be hard enough to understand what is in a database to start with, but if the data is bad, it will make your job orders of magnitude more difficult. If you have promised to do the work for $1,000 and it ends up taking 500 hours, you would have been better off with a spatula working midnights near the Eiffel Tower . . . in Paris, Tennessee.

How Much Is This Data Worth?

It's also important to place value judgments on data. In some cases, data will have great value in the monetary sense. For example, in the dental office example that will be presented later in Chapter 4, the value lies in the record of what has been done to the patient and how much has been billed to the patient and his or her insurance company. Without this documentation, digging out this data to eventually get paid for the work done might take hours and days. This data has a specific monetary value, because the quicker the payment is received, the more interest is drawn, meaning more profits. If the client shifts the turnover of payments from one month to one week because of streamlining the process, this might be worth quite a bit more money.

On the other hand, just because existing data is available doesn't necessarily mean that it should be included in the new database. The client needs to be informed of all the data that's available and should be provided with a cost estimate of transferring it into the new database. The cost of transferring legacy data can be high, and the client should be offered the opportunity to make decisions that might conserve funds for more important purposes.

Who Will Use the Data?

Who is going to use the data probably doesn't instantly jump out at you as a type of data that needs to be considered during the early stages of requirement gathering. When designing an interface, usually who is going to actually be pushing the button probably doesn't make a lot of difference to the button design (unless disabilities are involved in the equation perhaps). Yet, the answer to the question of "who" can start to answer several different types of questions:

- *Security*: "Who will use the data?" can be taken two ways. First, these are the only people who care about the data. Second, these are the only people who are privileged to use the data. The latter will require you to create boundaries to utilization. For fun, add in privacy laws like the Health Insurance Portability and Accountability Act (HIPAA) or Sarbanes–Oxley Act (SOX) or any of 100 other well-meaning laws that punish the dba more than the offenders.
- *Structure*: If multiple user groups need the same data, but for particularly different needs, this could lead to different possible design outcomes later in the process.

www.it-ebooks.info

- *Concurrency*: The design considerations for a system that has one user are different from ten, or a hundred, thousand, and so on. The number of users should not change our conceptual or logical designs, but it certainly can effect how we design the physical hardware layer. Concurrency is something we won't make a lot of reference to until very late in this book (Chapter 10, but this is the point in time when you are doing the asking (and likely purchasing the hardware, so it doesn't hurt to find out now).

This choice of who will use the data goes hand in hand with all of the other questions you have gotten answered during the process of gathering requirements. Of course, these questions are just the start of the information gathering process, but there is still a lot more work to go before you can start building a database, so you are going to have to cool your jets a bit longer.

Working with Existing Systems and Prototypes

If you're writing a new version of a current database system, access to the existing system is going to be a blessing *and* a curse. Obviously, the more information you can gather about how any previous system and its data was previously structured, the better. All the screens, data models, object models, user documents, and so on are important to the design process.

However, unless you're simply making revisions to an existing system, often the old database system is reasonable only as a reference point for completeness, not as an initial blueprint. On most occasions, the existing system you'll be replacing will have many problems that need to be fixed, not emulated. If the system being replaced had no problems, why is the client replacing it? Possibly just to more to newer technology but no one replaces a working system just for kicks.

Prototypes from the early design process might also exist. Prototypes can be useful tools to communicate how to solve a real-world problem using a computer or when you're trying to reengineer how a current process is managed. Their role is to be a proof of concept—an opportunity to flesh out with the design team and the end users the critical elements of the project on which success or failure will depend.

The real problem with prototypes is that if a database was created for the prototype, it is rarely going to be worth anything. So, by the time database design starts, you might be directed to take a prototype database that has been hastily developed and "make it work" or, worse yet, "polish it up." Indeed, you might inherit an unstructured, unorganized prototype, and your task will be to turn it into a production database in no time flat (loosely translated, that means to have it done early yesterday).

It may be up to you, at times, to remind customers to consider prototypes only as interactive pictures to get the customer to try out a concept, often to get your company to sign a contract. Sometimes, you might be hired to implement the prototype (or the failed try at a production system that's now being called a prototype) that another consultant was hired to create (or worse yet, an employee who still works there and has a chip on his or her shoulder the size of a large African elephant that spends all day eating bonbons and watching soap operas). It's better to start from scratch, developing the final application using structured, supported design and coding standards. As a data architect, you must work as hard as possible to use prototype code *only* as a working model—a piece of documentation that you use to enhance your own design. Prototypes help you to be sure you're not going to miss out on any critical pieces of information that the users need—such as a name field, a search operation, or even a button (which might imply a data element)—but they may not tell you anything about the eventual database design at all.

Utilizing Other Types of Documentation

Apart from interviews and existing systems, you can look to other sources to find data rules and other pieces of information relevant to the design project. Often, the project manager will obtain these documents; sometimes, they will not be available to you, and you just have to take someone else's word for what is in them. In these cases,

www.it-ebooks.info

I find it best to get into writing your understanding and make sure it is clear who said what about the meaning of documentation you cannot see. Ordinarily, I try not to worry about the later blame factor, but it is essential to worry when you are working from a translation that may later come under dispute.

And as always, the following list is certainly not exclusive but should kick start your thinking about where to get existing documentation for a system you are creating or replacing.

Early Project Documentation

If you work for a company that is creating software for other companies, you'll find that early in the project there are often documents that get created to solicit costs and possible solutions, for example:

- *Request for quote (RFQ)*: A document with a fairly mature specification that an organization sends out to determine how much a solution would cost
- *Request for proposal (RFP)*: For less mature ideas for which an organization wants to see potential solutions and get an idea about its costs

Each of these documents contains valuable information that can help you design a solution, because you can get an idea of what the client wanted before you got involved. Things change, of course, and not always will the final solution resemble the original request, but a copy of an RFP or an RFQ should be added to the pile of information that you'll have available later in the process. Although these documents generally consist of sketchy information about the problem and the desired solution, you can use them to confirm the original reason for wanting the database system and for getting a firmer handle on what types of data are to be stored within it.

No matter what, if you can get a copy of these documents, you'll be able to see the client's thought pattern and why the client wants a system developed.

Contracts or Client Work Orders

Getting copies of the contract can seem like a fairly radical approach to gathering design information, depending on the type of organization you're with. Frankly, in a corporate structure, you'll likely have to fight through layers of management to make them understand why you need to see the contract at all. Contracts can be inherently difficult to read because of the language they're written in (sort of like a terse version of a programming language, with intentional vagueness tossed in to give lawyers something to dispute with one another later). However, be diligent in filtering out the legalese, and you'll uncover what amounts to a basic set of requirements for the system—often the requirements that you must fulfill exactly or not get paid. Even more fun is the stuff you may learn that has been promised that the implementation team has never heard of.

What makes the contract so attractive is simple. It is, generally speaking, the target you'll be shooting at. No matter what the client says, or what the existing system was, if the contract specifies that you deliver some sort of watercraft and you deliver a Formula 1 race car because the lower-level clients change their minds without changing the contract, you might not get paid because your project is deemed a failure (figuratively speaking, of course, since who doesn't like Formula 1 race cars?).

Level of Service Agreement

One important section of contracts that's important to the design process is the required level of service. This might specify the number of pages per minute, the number of rows in the database, and so on. All this needs to be measured, stored, tested for, and so on. When it comes to the testing and optimization phases, knowing the target level of service can be of great value. You may also find some data that needs to be stored to validate that a service level is being met.

www.it-ebooks.info

Audit Plans

Don't forget about audits. When you build a system, you must consider whether the system is likely to be audited in the future and by whom. Government, ISO 9000 clients, and other clients that are monitored by standards organizations are likely to have strict audit requirements. Other clients may also have financial audit processes. Of particular concern are all the various privacy policies, child data restrictions, credit card encryption rules, and so on. All of these will not only require that you follow rules that regulatory bodies set but that you document certain parts of your operation. These audit plans might contain valuable information that can be used in the design process.

Following Best Practices

The following list of some best practices can be useful to follow when dealing with and gathering requirements:

- *Be diligent*: Look through everything to make sure that what's being said makes sense. Be certain to understand as many of the business rules that bind the system as possible before moving on to the next step. Mistakes made early in the process can mushroom later in the process.
- *Document*: The format of the documentation isn't really all that important, only that you get documented as much of what the client wants as possible. Make sure that the documentation is understandable by all parties involved and that it will be useful going forward toward implementation.
- *Communicate*: Constant communication with clients is essential to keep the design on track. The danger is that if you start to get the wrong idea of what the client needs, every decision past that point might be wrong. Get as much face time with the client as possible.

■ **Note** The mantra "review with client, review with client, review with client" is probably starting to get a bit old at this point. This is one of the last times I'll mention it in this chapter at least, but it's so important that I hope it has sunk in.

Summary

In this chapter, I've introduced some of the basics of documentation and requirements gathering. This is one of the most important parts of the process of creating software, because it's the foundation of everything that follows. If the foundation is solid, the rest of the process has a chance. If the foundation is shoddy, the rest of the system that gets built will likely be the same. The purpose of this process is to acquire as much information as possible about what the clients want out of their system. As a data architect, this information might be something that's delivered to you, or at least most of it. Either way, the goal is to understand the user's needs.

Once you have as much documentation as possible from the users, the real work begins. Through all this documentation, the goal is to prepare you for the next step of producing a data model that will document in a very formal manner of the following:

- Entities and relationships
- Attributes and domains

www.it-ebooks.info

- Business rules that can be enforced in the database
- Processes that require the use of the database

From this, a conceptual data model will emerge that has many of the characteristics that will exist in the actual implemented database. In the upcoming chapters, the database design will certainly change from this conceptual model, but it will share many of the same characteristics.

www.it-ebooks.info

CHAPTER 3

The Language of Data Modeling

The aim of art is to represent not the outward appearance of things but their inward significance.

—Aristotle

A data model is one of the most important tools in the design process, but it has to be done right. A common misconception is that a data model is a picture of a database. That is partly true, but a model can do so much more. A great data model covers pretty much everything about a database and serves as the primary documentation for the life cycle of the database. Aspects of the model will be useful to developers, users, and the database administrators (DBAs) who maintain the system.

In this chapter, I will introduce the basic concept of *data modeling*, in which a representation of your database will be produced that shows the objects involved in the database design and how they interrelate. It is really a description of the exterior and interior parts database, with a graphical representation being just one facet of the model (the graphical part of the model is probably the most interesting to a general audience, because it gives a very quick and easy-to-work-with overview of your objects and their relationships). Best of all, using a good tool, you can practically design the basics of a system live, right with your clients as they describe what they want (hopefully, someone else is gathering client requirements that are not data-structure related).

In the next section, I'll provide some basic information about data modeling and introduce the language I prefer for data modeling (and will use for many examples in this book): IDEF1X. I'll then cover how to use the IDEF1X methodology to model and document the following:

- Entities/tables
- Attributes/columns
- Relationships
- Descriptive information

In the process of creating a database, we will start out modeling entities and attributes, which do not follow very strict definitions, and refine the models until we end up producing tables and columns, which, as discussed in Chapter 1, have very formal definitions that we have started to define and will refine even more in Chapter 5. For this chapter and the next, we will primarily refer to entities during the modeling exercises, unless we're trying to demonstrate something that would be created in SQL Server. The same data modeling language will be used for the entire process of modeling the database, with some changes in terminology to describe an entity or a table later in this book.

www.it-ebooks.info

After introducing IDEF1X, we will briefly introduce several other alternative modeling methodology styles, including information engineering (also known as "crow's feet") and the Chen Entity Relationship Model (ERD) methodology. I'll also show an example of the diagramming capabilities built into SQL Server Management Studio.

■ **Note** This chapter will mainly cover the concepts of modeling. In the next chapter, we will apply these concepts to build a data model.

Introducing Data Modeling

Data modeling is a skill at the foundation of database design. In order to start designing databases, it is very useful to be able to effectively communicate the design as well as make it easier to visualize. Many of the concepts introduced in Chapter 1 have graphical representations that make it easy to get an overview of a vast amount of database structure and metadata in a very small amount of space. As mentioned earlier, a common misconception about the data model is that it is solely about painting a pretty picture. In fact, the model itself can exist without the graphical parts; it can consist of just textual information, and almost everything in the data model can be read in a manner that makes grammatical sense to almost any interested party. The graphical nature is simply there to fulfill the baking powder prophecy—that a picture is worth a thousand words. It is a bit of a stretch, because as you will see, the data model will have lots of words on it!

■ **Note** There are many types of models or diagrams: process models, data flow diagrams, data models, sequence diagrams, and others. For the purpose of database design, however, I will focus only on data models.

Several popular modeling languages are available to use, and each is generally just as good as the others at the job of documenting a database design. The major difference will be some of the symbology that is used to convey the information. When choosing my data modeling methodology, I looked for one that was easy to read and could display and store everything required to implement very complex systems. The modeling language I use is Integration Definition for Information Modeling (IDEF1X). (It didn't hurt that the organization I have worked for over ten years has used it for that amount of time too.)

IDEF1X is based on Federal Information Processing Standards Publication 184, published September 21, 1993. To be fair, the other major methodology, Information Engineering, is good too, but I like the way IDEF1X works, and it is based on a publicly available standard. IDEF1X was originally developed by the U.S. Air Force in 1985 to meet the following requirements:

- Support the development of data models.
- Be a language that is both easy to learn and robust.
- Be teachable.
- Be well tested and proven.
- Be suitable for automation.

www.it-ebooks.info

■ **Note** At the time of this writing, the full specification for IDEF1X is available at http://www.itl.nist.gov/fipspubs/idef1x.doc. The exact URL of this specification is subject to change, but you can likely locate it by searching the http://www.itl.nist.gov site for "IDEF1X."

While the selection of a data modeling methodology may be a personal choice, economics, company standards, or features usually influence tool choice. IDEF1X is implemented in many of the popular design tools, such as the following, which are just a few of the products available that claim to support IDEF1X (note that the URLs listed here were correct at the time of this writing, but are subject to change in the future):

- *AllFusion ERwin Data Modeler*: http://erwin.com/products/detail/ca_erwin_data_modeler/
- *Toad Data Modeler*: http://www.quest.com/toad-data-modeler/
- *ER/Studio*: http://www.embarcadero.com/products/er-studio-xe
- *Visible Analyst DB Engineer*: http://www.visible.com/Products/Analyst/vadbengineer.htm
- *Visio Enterprise Edition*: http://www.microsoft.com/office/visio

Let's next move on to practice modeling and documenting, starting with entities.

Entities

In the IDEF1X standard, entities (which, as discussed previously, are loosely synonymous with tables) are modeled as rectangular boxes, as they are in most data modeling methodologies. Two types of entities can be modeled: identifier-independent and identifier-dependent, usually referred to as "independent" and "dependent," respectively.

The difference between a dependent entity and an independent entity lies in how the primary key of the entity is structured. The independent entity is so named because it has no primary key dependencies on any other entity, or in other words, the primary key contains no foreign key columns from other entities.

Chapter 1 introduced the term "foreign key," and the IDEF1X specification introduces an additional term: *migrated*. A foreign key is referred to as a migrated key when the key of a parent table is moved into the child table. The term "migrated" can be slightly misleading, because the primary key of one entity is not actually moving; rather, in this context, the primary key of one entity is copied as an attribute in a different entity to establish a relationship between the two entities. However, knowing its meaning in this context (and a slight release of your data-architect anal-retentive behavior), "migrated" is a good term to indicate what occurs when you put the primary key of one entity into another table to set up the reference.

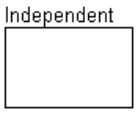
If the primary key of one entity is migrated into the primary key of another, it is considered dependent on the other entity, because one entity's meaning depends on the existence of the other. If the attributes are migrated to the nonprimary key attributes, they are independent of any other entities. All attributes that are not migrated as foreign keys from other entities are *owned*, as they have their origins in the current entity. Other methodologies and tools may use the terms "identifying" and "nonidentifying" instead of "owned" and "independent."

For example, consider an invoice that has one or more line items. The primary key of the invoice entity might be `invoiceNumber`. If the invoice has two line items, a reasonable choice for the primary key would be `invoiceNumber` and `lineNumber`. Since the primary key contains `invoiceNumber`, it would be dependent on the `invoice` entity. If you had an `invoiceStatus` entity that was also related to `invoice`, it would be independent,

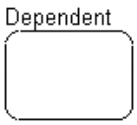
www.it-ebooks.info

because an invoice's existence is not really predicated on the existence of a status (even if a value for the `invoiceStatus` to `invoice` relationship *is* required (in other words, the foreign key column would be `NOT NULL`).

An independent entity is drawn with square corners, as follows:



The dependent entity is the converse of the independent entity—it will have the primary key of one or more entities migrated into its primary key. It is called "dependent" because its identifier depends on the existence of another entity. It is drawn with rounded corners, as follows:



■ **Note** The concept of dependent and independent entities lead us to a bit of a chicken and egg paradox (not to mention, a fork in the road). The dependent entity is dependent on a certain type of relationship. However, the introduction of entity creation can't wait until after the relationships are determined, since the relationships couldn't exist without entities. If this is the first time you've looked at data models, this chapter may require a reread to get the full picture, as the concept of independent and dependent objects is linked to relationships.

As we start to identify entities, we need to deal with the topic of naming. One of the most important aspects of designing or implementing any system is how objects, variables, and so forth are named. Long discussions about names always seem like a waste of time, but if you have ever gone back to work on code that you wrote months ago, you understand what I mean. For example, `@x` might seem like an OK variable name when you first write some code, and it certainly saves a lot of keystrokes versus typing `@holdEmployeeNameForCleaningInvalidCharacters`, but the latter is much easier to understand after a period of time has passed (for me, this period of time is approximately 14.5 seconds).

Naming database objects is no different; actually, naming database objects clearly is more important than naming other programming objects, as your end users will almost certainly get used to these names: the names given to entities will be translated into table names that will be accessed by programmers and users alike. The conceptual and logical model will be considered your primary schematic of the data in the database and should be a living document that you change before changing any implemented structures.

Frequently, discussions on how objects should be named can get heated because there are several different schools of thought about how to name objects. The central issue is whether to use plural or singular names. Both have merit, but one style has to be chosen. I choose to follow the IDEF1X standard for object names, which says to use singular names. By this standard, the name itself doesn't name the container but, instead, refers to an instance of what is being modeled. Other standards use the table's name for the container/set of rows.

www.it-ebooks.info

Is either way more correct? Each has benefits; for example, in IDEF1X, singular entity/table names lead to the ability to read the names of relationships naturally. But honestly, plural or singular naming might be worth a few long discussions with fellow architects, but it is certainly not something to get burned at the stake over. If the organization you find yourself beholden to uses plural names, that doesn't make it a bad place to work. The most important thing is to be consistent and not let your style go all higgledy-piggledy as you go along. Even a bad set of naming standards is better than no standards at all, so if the databases you inherit use plural names, follow the "when in Rome" principle and use plural names so as not to confuse anyone else.

In this book, I will follow these basic guidelines for naming entities:

- *Entity names should never be plural*. The primary reason for this is that the name should refer to an instance of the object being modeled, rather than the collection. This allows you to easily use the name in a sentence. It is uncomfortable to say that you have an "automobiles row," for example—you have an "automobile row." If you had two of these, you would have two automobile rows.
- *The name given should directly correspond to the essence of what the entity is modeling*. For instance, if you are modeling a person, name the entity `Person`. If you are modeling an automobile, call it `Automobile`. Naming is not always this straightforward, but keeping the name simple and to the point is wise. If you need to be more specific, that is fine too. Just keep it succinct (unlike this explanation!).

Entity names frequently need to be made up of several words. During the conceptual and logical modeling phases, including spaces, underscores, and other characters when multiple words are necessary in the name is acceptable but not required. For example, an entity that stores a person's addresses might be named `Person Address`, `Person_Address`, or using the style I have recently become accustomed to and the one I'll use in this book, `PersonAddress`. This type of naming is known as *Pascal case* or *mixed case*. (When you don't capitalize the first letter, but capitalize the first letter of the second word, this style is known as *camelCase*.) Just as in the plural/singular argument, there really is no "correct" way; these are just the guidelines that I will follow to keep everything uniform.

Regardless of any style choices you make, very few abbreviations should be used in the logical naming of entities unless it is a universal abbreviation that every person reading your model will know. Every word ought to be fully spelled out, because abbreviations lower the value of the names as documentation and tend to cause confusion. Abbreviations may be necessary in the implemented model because of some naming standard that is forced on you or a very common industry standard term. Be careful of assuming the industry-standard terms are universally known. For example, at the time of this writing, I am helping breaking in a new developer at work, and every few minutes, he asks what a term means—and the terms are industry standard.

If you decide to use abbreviations in any of your names, make sure that you have a standard in place to ensure the same abbreviation is used every time. One of the primary reasons to avoid abbreviations is so you don't have to worry about different people using `Description`, `Descry`, `Desc`, `Descrip`, and `Descriptn` for the same attribute on different entities.

Often, novice database designers (particularly those who come from interpretive or procedural programming backgrounds) feel the need to use a form of *Hungarian notation* and include prefixes or suffixes in names to indicate the kind of object—for example, `tblEmployee` or `tblCustomer`. Prefixes like this are generally considered a bad practice, because names in relational databases are almost always used in an obvious context. Using Hungarian notation is a good idea when writing procedural code (like Visual Basic or C#), since objects don't always have a very strict contextual meaning that can be seen immediately upon usage, especially if you are implementing one interface with many different types of objects. In SQL Server Integration Services (SSIS) packages, I commonly name each control with a three- or four-letter prefix to help identify them in logs. However, with database objects, questioning whether a name refers to a column or a table is rare. Plus, if the object type isn't obvious, querying the system catalog to determine it is easy. I won't go too far into implementation right now, but you can use the `sys.objects` catalog view to see the type of any object. For example, this query will

www.it-ebooks.info

list all of the different object types in the catalog (your results may vary; this query was executed against the `AdventureWorks2012` database we will use for some of the examples in this book):

```
SELECT DISTINCT type_desc
FROM sys.objects
```

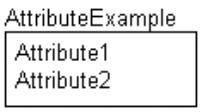
Here's the result:

```
type_desc
------------------------
CHECK_CONSTRAINT
DEFAULT_CONSTRAINT
FOREIGN_KEY_CONSTRAINT
INTERNAL_TABLE
PRIMARY_KEY_CONSTRAINT
SERVICE_QUEUE
SQL_SCALAR_FUNCTION
SQL_STORED_PROCEDURE
SQL_TABLE_VALUED_FUNCTION
SQL_TRIGGER
SYNONYM
SYSTEM_TABLE
UNIQUE_CONSTRAINT
USER_TABLE
VIEW
```

We will use `sys.objects` and other catalog views throughout this book to view properties of objects that we create.

Attributes

All attributes in the entity must be uniquely named within it. They are represented by a list of names inside of the entity rectangle:

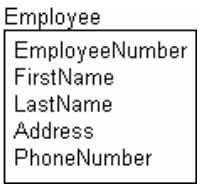


■ **Note** The preceding image shows a technically invalid entity, as there is no primary key defined (a requirement of IDEF1X). I'll cover the notation for keys in the following section.

At this point, you would simply enter all of the attributes that you discover from the requirements (the next chapter will demonstrate this process). In practice, you would likely have combined the process of discovering entities and attributes with the initial modeling phase (we will do so in Chapter 4 as we go through the process of creating a logical data model). Your process will depend on how well the tools you use work. Most data modeling tools cater for building models fast and storing a wealth of information along the way to document their entities and attributes.

www.it-ebooks.info

In the early stages of logical modeling, there can be quite a large difference between an attribute and what will be implemented as a column. As I will demonstrate in Chapter 5, the attributes will be transformed a great deal during the normalization process. For example, the attributes of an `Employee` entity may start out as follows:



However, during the normalization process, tables like this will often be broken down into many attributes (e.g., `address` might be broken into `number`, `street name`, `city`, `state`, `zip code`, etc.) and possibly many different entities.

■ **Note** Attribute naming is one place where I tend to deviate from IDEF1X standard. The standard is that names are unique within a model. This tends to produce names that include the table name followed by the attribute name, which can result in unwieldy, long names that look archaic. You can follow many naming standards you can follow to avoid unwieldy names (and even if I don't particularly like them), some with specific abbreviations, name formats, and so forth. For example, a common one has each name formed by a descriptive name and a class word, which is an abbreviation like `EmployeeNumber`, `ShipDate`, or `HouseDescription`. For sake of nonpartisan naming politics, I am happy to say that any decent naming standard is acceptable, as long as it is followed.

Just as with entity names, there is no need to include Hungarian notation prefixes or suffixes in the attribute or implementation names. The type of the attribute can be retrieved from the system catalog if there is any question about it.

Next, we will go over the following aspects of attributes on your data model:

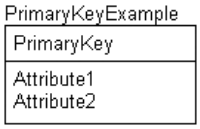
- Primary keys
- Alternate keys
- Foreign keys
- Domains
- Attribute naming

Primary Keys

As noted in the previous section, an IDEF1X entity must have a primary key. This is convenient for us, because an entity is defined such that each instance must be unique (see Chapter 1). The primary key may be a single attribute, or it may be a composite of multiple attributes. A value is required for every attribute in the key (logically speaking, no `NULL`s are allowed in the primary key).

www.it-ebooks.info

The primary key is denoted by placing attributes above a horizontal line through the entity rectangle. Note that no additional notation is required to indicate that the value is the primary key.



For example, consider the `Employee` entity from the previous section. The `EmployeeNumber` attribute is unique, and logically, every employee would have one, so this would be an acceptable primary key:



The choice of primary key is an interesting one. In the early logical modeling phase, I generally do not like to spend time choosing the final primary key attribute(s). The main reason for this is to avoid worrying too much about what the key is going to be. I tend to create a simple surrogate primary key to migrate to other entities to help me see when there is any ownership. In the current example, `EmployeeNumber` clearly refers to an employee, but not every entity will be so clear—not to mention that more advanced business rules may dictate that `EmployeeNumber` is not always unique. (For example, the company also may have contractors in the table. That's not a good practice perhaps, but no matter how much I try to describe perfect databases in this book, not every table will end up being perfect.) Having to repeatedly go back and change the entity used for the primary key in the logical model over and over can be tiresome, particularly when you have a very large model.

It is also quite likely that you may have multiple column sets that uniquely identify a given instance of many of your entities. As an example, consider an entity that models a product manufactured by a company. The company may identify the product by the type, style, size, and series:

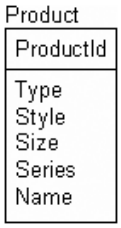


The name may also be a good key, and more than likely, there is also a product code. Which attribute is the best key—or which is even truly a key—may not become completely apparent until later in the process. There are many ways to implement a good key, and the best way may not be recognizable right away.

Instead of choosing a primary key at this point, I add a value to the entity for identification purposes and then model all candidate keys as alternate keys (which I will discuss in the next section). As a result, the logical

www.it-ebooks.info

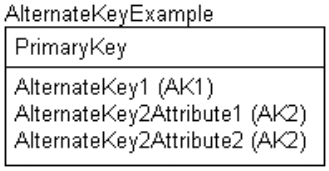
model clearly shows what entities are in an ownership role to other entities, since the key that is migrated contains the name of the modeled entity. I would model this entity as follows:



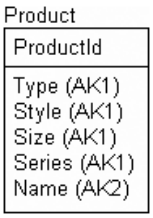
■ **Note** Using surrogate keys is certainly not a requirement in logical modeling; it is a personal preference that I have found a useful documentation method to keep models clean, and it corresponds to my method of implementation later. Not only is using a natural key as the primary key in the logical modeling phase reasonable but many architects find it preferable. Either method is perfectly acceptable (and just as likely to start a religious debate at a table of data modelers. You have been warned, so start the debate after the desert course).

Alternate Keys

As defined in Chapter 1, an alternate key is a grouping of one or more attributes whose uniqueness needs to be guaranteed over all of the instances of the entity. Alternate keys do not have specific locations in the entity graphic like primary keys, nor are they typically migrated for any relationship (you can reference an alternate key with a foreign key based on the SQL standards, but this feature is very rarely used feature, and when used, it will often really confuse even the best DBAs). They are identified on the model in a very simple manner:

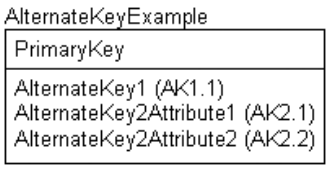


In this example, there are two alternate *key groups*: group `AK1`, which has one attribute as a member, and group `AK2`, which has two attributes. There also is nothing wrong with overlapping alternate keys, which could be denoted as `(AK1,AK2)`. Thinking back to the product example, the two keys could then be modeled as follows:



www.it-ebooks.info

One extension that Computer Associates' ERwin adds to this notation is shown here:

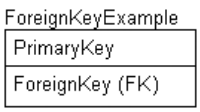


A position number notation is tacked onto the name of each key (`AK1` and `AK2`) to denote the position of the attribute in the key. In the logical model, technically, the order of attributes in the key should not be considered even if the tool does display them (unique is unique, regardless of key column order). Which attribute comes first in the key really does not matter; all that] matters is that you make sure there are unique values across multiple attributes. When a key is implemented, the order of columns *will* become interesting for performance reasons (because SQL Server implements uniqueness with an index), but uniqueness will be served no matter what the order of the columns of the key is.

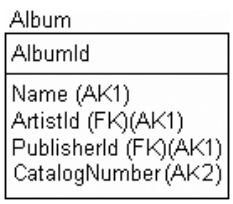
■ **Note** The discussion of index utilization for performance reasons is left to Chapter 10. Do your best to more or less ignore performance tuning needs during the conceptual, logical, and even most of the physical modeling phases. Defer performance tuning issues until the storage modeling phase, which includes query tuning, indexing, and disk layout.

Foreign Keys

Foreign key attributes, as I've alluded to, are also referred to as migrated attributes. They are primary keys from one entity that serve as references to an instance in another entity. They are, again, a result of relationships (we'll look at their graphical representation later in this chapter). They are indicated, much like alternate keys, by adding the letters "FK" after the foreign key:



As an example of a table with foreign keys, consider an entity that is modeling a music album:



The `artistId` and `publisherId` represent migrated foreign keys from the artist and publisher entities. We'll revisit this example in the "Relationships" section later in this chapter.

www.it-ebooks.info

One tricky thing about this example is that the diagram doesn't show what entity the key is migrated from. This can tend to make things a little messy, depending on how you choose your primary keys. This lack of clarity about what table a foreign key migrates from is a limitation of most modeling methodologies, because displaying the name of the entity where the key came from would be unnecessarily confusing for a couple of reasons:

- There is no limit (nor should there be) on how far a key will migrate from its original owner entity (the entity where the key value was not a migrated foreign key reference).
- It is not completely unreasonable that the same attribute might migrate from two separate entities with the same name, especially early in the logical design process. This is certainly not a design goal, but it is possible and can make for interesting situations.

One of the reasons for the primary key scheme I will employ in logical models is to add a key named `<entityName>Id` as the identifier for entities, so the name of the entity is easily identifiable and lets us easily know where the original source of the attribute is. Also, we can see the attribute migrated from entity to entity even without any additional documentation. For example, in the `Album` entity example, we instinctively know that the `ArtistId` attribute is a foreign key and most likely was migrated from the `Artist` entity just because of the name alone.

Domains

In Chapter 1, the term "domain" referred to a set of valid values for an attribute. In IDEF1X, you can formalize domains and define named, reusable specifications known as domains, for example:

- `String`: A character string
- `SocialSecurityNumber`: A character value with a format of `###-##-####`
- `PositiveInteger`: An integer value with an implied domain of `0` to `max(integer value)`
- `Truth`: A five-character value with a domain of `('FALSE','TRUE')`

Domains in the specification not only allow us to define the valid values that can be stored in an attribute but also provide a form of inheritance in the datatype definitions. *Subclasses* can then be defined of the domains that inherit the settings from the base domain. It is a good practice to build domains for any attributes that get used regularly, as well as domains that are base templates for infrequently used attributes. For example, you might have a character type domain where you specify a basic length, like 60. Then, you may specify common domains, like name and description, to use in many entities. For these, you should choose a reasonable length for the values, plus you could include requirements that the data in the column cannot be just space characters to prevent a user from having one, two, or three spaces each look like different values—except in the rare cases where that is desirable.

Regardless of whether or not you are using an automated tool for modeling, try to define common domains that you use for specific types of things (applying a common pattern to solve a common problem). For example, a person's first name might be a domain. This is cool because you don't have to answer "Hmm, how long to make a person's name?" or "what is the format of our part numbers?" or any similar questions more than once. After you make a decision, you just use what you have used before.

■ **Note** Defining common domains during design fights against `varchar(200)` syndrome, where every column in a database stores textual data in columns of exactly same length. Putting in some early thought on the minimum and maximum lengths of data is easier than doing it when the project manager is screaming for results later in the process, and the programmers are champing at the bit to get at your database and get coding.

---

www.it-ebooks.info

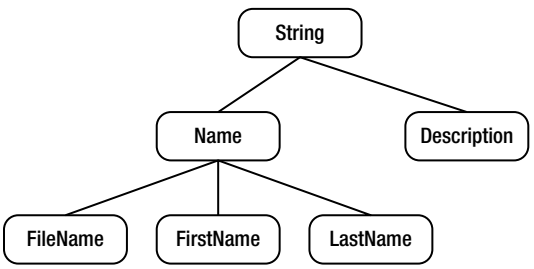
Early in the modeling process, you'll commonly want to gather a few bits of information, such as the general type of the attribute: character, numeric, logical, or even binary data. Determining minimum and maximum lengths may or may not be possible, but the more information you can gather without crushing the process the better. Another good thing to start is documenting the legal values for an attribute that is classified as being of the domain type. This is generally done using some pseudocode or in a textual manner, either in your modeling tool or even in a spreadsheet.

It is extremely important to keep these domains as implementation-independent datatype descriptions. For example, you might specify a domain of `GloballyUniqueIdentifier`, a value that will be unique no matter where it is generated. In SQL Server, a unique identifier could be used (GUID value) to implement this domain. In another operating system (created by a company other than Microsoft, perhaps) where there is not exactly the same mechanism, it might be implemented differently; the point is that this value is statistically guaranteed to be unique every time it is generated. The conceptual/logical modeling phase should be done without too much thinking about what SQL Server can do, if for no other reason than to prevent you from starting to impose limitations on the future solution prior to understanding the actual problem. Another sort of domain might be a set of legal values, like if the business users had defined three customer types, you could specify the legal string values that could be used.

When you start the physical modeling of the relational structures, you will use the same domains to assign the implementation properties. This is the real value in using domains. By creating reusable template attributes that will also be used when you start creating columns, you'll spend less effort and time building simple entities, which make up the bulk of your work. Doing so also provides a way for you to enforce companywide standards, by reusing the same domains on all corporate models (predicated, of course, on you being diligent with your data modeling processes over time!).

Later on, implementation details such as exact datatypes, constraints, and so forth will be chosen, just to name a few of the more basic properties that may be inherited (and if Microsoft adds a better datatype in the future, you can simply change all of the columns with that domain type to the new type). Since it is very likely that you will have fewer domains than implemented attributes, the double benefit of speedy and consistent model assembly is achieved. However, it is probably not overly reasonable or even useful to employ the inheritance mechanisms when building tables by hand. Implementation of a flat domain structure is enough work without a tool.

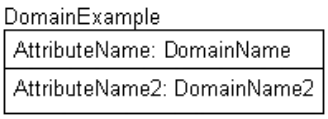
As an example of a domain hierarchy, consider this set of character string domains:



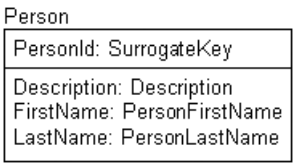
Here, `String` is the base domain from which you can then inherit `Name` and `Description`. `FileName`, `FirstName`, and `LastName` are inherited from `Name`. During logical modeling, this might seem like a lot of work for nothing, because most of these domains will share only a few basic details, such as not allowing `NULL`s or blank data. However, `FileName` may be optional, whereas `LastName` might be mandatory. Setting up domains for as many distinct attribute types as possible is important, in case rules or datatypes are discovered that are common to any domains that already exist. Things get good when you need to change all of your datatypes for all string types, for example, if you decide to make a blanket change from ANSI character sets to UNICODE, or to implement compression on all description type attributes but not name ones.

www.it-ebooks.info

Domains are a nice feature of IDEF1X (and other methodologies or tools that support them). They provide an easy method of building standard attribute types, reducing both the length of time required for repeating common attribute types and the number of errors that occur in doing so. Specific tools implement domains with the ability to define and inherit more properties throughout the domain chain to make creating databases easier. During logical modeling, domains might optionally be shown to the right of the attribute name in the entity:



So if I have an entity that holds domain values for describing a type of person, I might model it as follows:



To model this example, I defined four domains:

- `SurrogateKey`: The surrogate key value. (Implementation of the surrogate should not be implied by building a domain, so later, this can be implemented in any manner.) I could also choose to use a natural key.
- `Description`: Holds the description of "something" (can be 60 characters maximum).
- `PersonFirstName`: A person's first name (30 characters maximum).
- `PersonLastName`: A person's last name (50 characters maximum).

The choice of the length of name is an interesting one. I searched on Google for "person first name varchar" and found lots of different possibilities: 10, 35, unlimited, 25, 20, and 15— all on the first page of the search! Just as you should use a consistent naming standard, you should use standard lengths every time like data is represented, so when you hit implementation, the likelihood that two columns storing like data will have different definitions is minimized.

During the implementation phase, all of the domains will get mapped to some form of datatype or, if you are so inclined, a user-defined type in SQL Server. However, the future implementation isn't quite the point at this point of the process. The point of a domain in the logical model is to define common types of storage patterns that can be applied in a common manner, including all of the business rules that will govern their usage.

Naming

Attribute naming is a bit more interesting than entity naming. I stated earlier that my preference is to use singular, not plural, entity names. The same issues that apply in entity naming are technically true for attribute naming (and no one disagrees with this!). However, until the logical model is completed, the model may still have attribute names that are plural. Leaving a name plural during early modeling phases can be a good reminder that you expect multiple values, but in my final relational table model, all attributes are almost always singular. For example, consider a `Person` entity with a `Children` attribute identified. The `Person` entity would identify a single person, and the `Children` attribute would identify sons and daughters of that person.

www.it-ebooks.info

The naming standard I follow is very straightforward and is intended to guide the names without being overly specific:

- Avoid repeating the entity name in the attribute name for most attributes except where it is natural to do so. The most common place where we include the table name is with some primary key attributes, particularly surrogate keys, since the key is specific for that table and some code values that are used in common language. For attributes that are not migrated to other entities, there is no need to prefix the name with the entity.
- Choose an attribute name to reflect precisely what is contained in the attribute and how it relates to the entity.
- Use abbreviations rarely in attribute names, especially for the conceptual/logical model. Every word ought to be spelled out in its entirety. If, for some reason, an abbreviation must be used (for example, due to the naming standard currently in use), a method should be put into place to make sure the abbreviation is used consistently, as discussed earlier in this chapter. For example, if your organization has a ZRF "thing" that is commonly used and referred to in general conversation as a ZRF, you might use this abbreviation. In general, however, I recommend avoiding abbreviations in all naming unless the client is insistent.
- Include no information in the name other than that necessary to explain the meaning of the attribute. This means no Hungarian notation of the type of data it represents (e.g., `LastNameString`) or prefix notation to tell you that it is, in fact, an attribute.
- End the name with a suffix that denotes general usage (often called a *classword*). It helps to standardize names and to let the user know the purpose of the column. Examples are:
  - `UserName`: A textual string referencing the name of the user. Whether or not it is a `varchar(30)` or `nvarchar(128)` is immaterial.
  - `PaymentId`: An identifier for a payment, usually a surrogate key. It can be implemented using a GUID, an integer, a random set of characters, and so on.
  - `EndDate`: The date when something ends. It does not include the time.
  - `SaveTime`: The point in time when the row was saved.
  - `PledgeAmount`: An amount of money (using a `numeric(12,2)`, or `money`, or any sort of type).
  - `DistributionDescription`: A textual string that is used to describe how funds are distributed.
  - `TickerCode`: A short textual string used to identify a ticker row.

■ **Note** Attribute names in the finalized logical and implementation models will not be plural, but we'll work this out in Chapter 5 when normalizing the model. At this point, it is not a big deal at all and actually is desirable if the values represented by a column are plural (and not simple atomic values as discussed in Chapter 1). Implementing this way would be bad, but during design, the only time it matters that the design is perfect is when it is being used for production.

---

www.it-ebooks.info

As in pretty much all things, consistency is the key to proper naming, so if you or your organization does not have a standard naming policy, developing one is worthwhile, even if it is very simple in nature. The overarching principle of my naming philosophy is to keep it simple and readable and to avoid all but universally standard corporate abbreviations. This standard will be followed from logical modeling into the implementation phase. Whatever your standard is, establishing a pattern of naming will make your models easy to follow, both for yourself and for your programmers and users. As always, any standard is better than no standard, and sometimes just the statement "make new columns and tables use the same names as those around them" is a good enough start if you have existing systems you are modifying. Trying to enforce a new standard on new work in an old system can make life more complicated than it is worth.

Relationships

Up to this point, the constructs we have looked at have been pretty much the same across most data modeling methodologies. Entities are always signified by rectangles, and attributes are quite often words within the rectangles. Relationships are where things start to diverge greatly, as many of the different modeling languages approach representing relationships graphically a bit differently. To make the concept of relationships clear, I need to go back to the terms "parent" and "child." Consider the following definitions from the IDEF1X specification's glossary (as these are remarkably lucid definitions to have been taken straight from a government specification!):

- *Entity, Child*: The entity in a specific connection relationship whose instances can be related to zero or one instance of the other entity (parent entity)
- *Entity, Parent*: An entity in a specific connection relationship whose instances can be related to a number of instances of another entity (child entity)
- *Relationship*: An association between two entities or between instances of the same entity

In IDEF1X, every relationship is denoted by a line drawn between two entities, with a solid circle at one end of that line to indicate where the primary key attribute is migrated to as a foreign key. In the following image, the primary key of the parent will be migrated to the child. This is how to denote a foreign key on a model.



Relationships come in several different flavors that indicate *how* the parent table is related to the child. We will look at examples of several different types of relationships in this section:

- *Identifying*, where the primary key of one table is migrated to the primary key of another. The child will be a dependent entity.
- *Nonidentifying*, where the primary key of one table is migrated to the nonprimary key attributes of another. The child will be an independent entity as long as no identifying relationships exist.
- *Optional identifying*, when the nonidentifying relationship does not require a child value.
- *Recursive relationships*, when a table is related to itself.
- *Subtype* or *categorization*, which is a one-to-one relationship used to let one entity extend another.
- *Many-to-many*, where an instance of an entity can be related to many in another, and in turn, many instances of the second entity can be related to multiples in the other.

www.it-ebooks.info

We'll also cover the *cardinality* of the relationship (how many of the parent relate to how many of the child), *role names* (changing the name of a key in a relationship), and *verb phrases* (the name of the relationship).

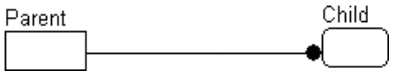
Relationships are a key topic in database design but not a completely simple one. A lot of information is related using a few dots and lines.

■ **Note** All of the relationships discussed in this section (except many-to-many) are of the one-to-many variety, which encompasses one-to-zero, one-to-one, one-to-many, or perhaps exactly-*n* relationships. Technically, it is more accurately one-to-(from M to N), as this enables specification of the many in very precise (or very loose) terms as the situation dictates. However, the more standard term is "one-to-many," and I will not try to make an already confusing term more so.

Identifying Relationships

The concept of a relationship being *identifying* is used to indicate containership, that the essence (defined as the intrinsic or indispensable properties that serve to characterize or identify something) of the child instance is defined by the existence of a parent. Another way to look at this is that generally the child in an identifying relationship is an inseparable part of the parent. Without the existence of the parent, the child would make no sense.

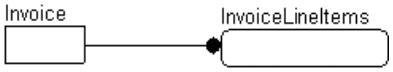
The relationship is drawn as follows:



To implement this relationship in the model, the primary key attribute(s) are migrated to the primary key of the child. Because of this, the key of a parent instance is needed to be able to identify a child instance record, which is why the name "identifying relationship" is used. In the following example, you can see that the `ParentId` attribute is a foreign key in the `Child` entity, from the `Parent` entity.



The child entity in the relationship is drawn as a rounded-off rectangle, which, as mentioned earlier in this chapter, means it is a dependent entity. A common example is an invoice and the line items being charged to the customer on the invoice:



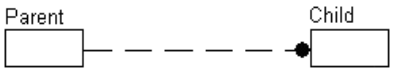
Without the existence of the invoice, the line items would have no purpose to exist. It can also be said that the line items are identified as being part of the parent.

www.it-ebooks.info

Nonidentifying Relationships

In contrast to identifying relationships, where relationships indicated that the child was an essential part of the parent entity, the *nonidentifying relationship* indicates that the child represents a more informational attribute of the parent.

When implementing the nonidentifying relationship, the primary key attribute is not migrated to the primary key of the child. It is denoted by a dashed line between the entities. Note too that the rectangle representing the child now has squared off corners, since it stands alone, rather than being dependent on the `Parent`:



This time, the key attribute of the parent will not migrate to the primary key of the child entity; instead, it will be in the nonprimary-key attributes.

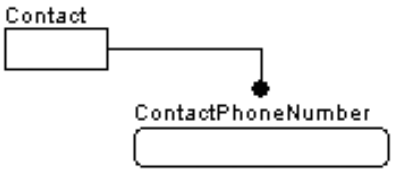


Taking again the example of an invoice, consider the vendor of the products that have been sold and documented as such in the line items. The product vendor does not define the existence of a line item, because with or without specifying the exact vendor the product originates from, the line item still makes sense.

The difference between identifying and nonidentifying relationships can be tricky but is essential to understanding the relationship between tables and their keys. If the parent entity defines the need for the existence of the child (as stated in the previous section), then use an identifying relationship. If, on the other hand, the relationship defines one of the child's attributes, use a nonidentifying relationship.

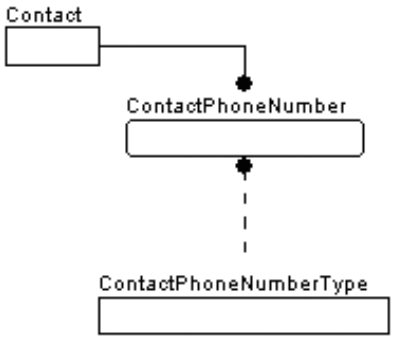
Here are some examples:

- *Identifying*: You have an entity that stores a contact and another that stores the contact's telephone number. `Contact` defines the phone number, and without the contact, there would be no need for the `ContactPhoneNumber`.



- *Nonidentifying*: Consider the entities that were defined for the identifying relationship, along with an additional entity called `ContactPhoneNumberType`. This entity is related to the `ContactPhoneNumber` entity, but in a nonidentifying way, and defines a set of possible phone number types (`Voice`, `Fax`, etc.) that a `ContactPhoneNumber` might be. The type of phone number does not identify the phone number; it simply classifies it. Even if the type wasn't known, recording the phone number could still be valid, as the number still has informational merit. However, a row associating a contact with a phone number would be useless information without the contact's existence.

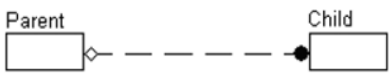
www.it-ebooks.info



The `ContactPhoneNumberType` entity is commonly known as a *domain entity* or *domain table*, as it serves to implement an attributes domain in a nonspecific manner. Rather than having a fixed domain for an attribute, an entity is designed that allows programmatic changes to the domain with no recoding of constraints or client code. As an added bonus, you can add columns to define, describe, and extend the domain values to implement business rules. It also allows the client user to build lists for users to choose values with very little programming.

While every nonidentifying relationship defines the domain of an attribute of the child table, sometimes when the row is created, the values don't need to be selected. For example, consider a database where you model houses, like for a neighborhood. Every house would have a color, a style, and so forth. However, not every house would have an alarm company, a mortgage holder, and so on. The relationship between the alarm company and bank would be optional in this case, while the color and style relationships would be mandatory. The difference in the implemented table will be whether or not the child table's foreign key will allow nulls. If a value is required, then it is considered *mandatory*. If a value of the migrated key can be null, then it is considered *optional*.

The optional case is signified by an open diamond at the opposite end of the dashed line from the black circle, as shown here:



In the mandatory case, the relationship is drawn as before, without the diamond. Note that an optional relationship means that the cardinality of the relationship may be zero, but a mandatory relationship must have a cardinality of one or greater (*cardinality* refers to the number of values that can be related to another value, and the concept will be discussed further in the next section).

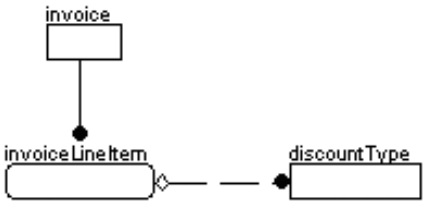
So why would you make a relationship optional? Consider once again the nonidentifying relationship between the invoice line item and the product vendor. The vendor in this case may be required or not required as the business rules dictate. If it is not required, you should make the relationship optional.

■ **Note** You might be wondering why there is not an optional identifying relationship. This is because you may not have any optional attributes in a primary key, which is true in relational theory and for SQL Server as well.

---

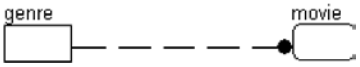
www.it-ebooks.info

For a one-to-many, optional relationship, consider the following:



The `invoiceLineItem` entity is where items are placed onto an invoice to receive payment. The user may sometimes apply a standard discount amount to the line item. The relationship, then, from the `invoiceLineItem` to the `discountType` entity is an optional one, as no discount may have been applied to the line item.

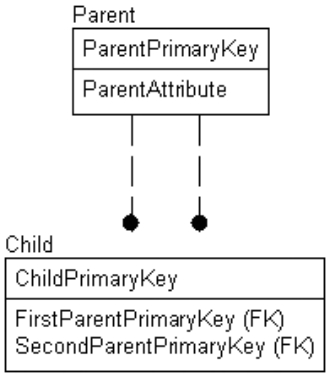
For most optional relationships like this, there is another possible solution, which can be modeled as required, and in the implementation, a row can be added to the `discountType` table that indicates "none." An example of such a mandatory relationship could be genre to movie in a movie rental systemdatabase:



The relationship is `genre<classifies>movie`, where the `genre` entity represents the "one" and `movie` represents the "many" in the one-to-many relationship. Every movie being rented must have a genre, so that it can be organized in the inventory and then placed on the appropriate rental shelf.

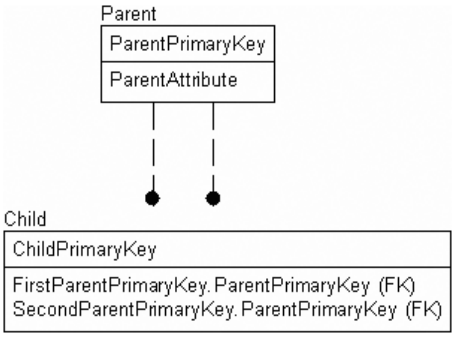
Role Names

A *role name* is an alternative name you can give an attribute when it is used as a foreign key. The purpose of a role name is to clarify the usage of a migrated key, because either the parent entity is generic and a more specific name is needed or the same entity has multiple relationships. As attribute names must be unique, assigning different names for the child foreign key references is often necessary. Consider these tables:

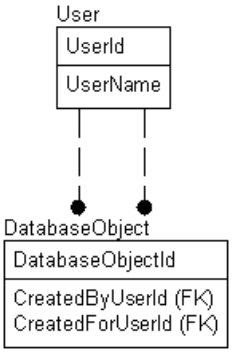


www.it-ebooks.info

In this diagram, the `Parent` and `Child` entities share two relationships, and the migrated attributes have been role named as `FirstParentPrimaryKey` and `SecondParentPrimaryKey`. In diagrams, you can indicate the original name of the migrated attribute after the role name, separated by a period (`.`), as follows (but usually it takes up too much space on the model when you are using it):



As an example, say you have a `User` entity, and you want to store the name or ID of the user who created a `DatabaseObject` entity instance as well as the user that the `DatabaseObject` instance was created for. It would then end up as follows:



Note that there are two relationships to the `DatabaseObject` entity from the `User` entity. Due to the way the lines are drawn on a diagram, it is not clear from the diagram which foreign key goes to which relationship. Once you name the relationship (with a verb phrase, which will be covered later in this chapter), the keys' relationships will be easier to determine, but often, determining which line indicates which child attribute is simply trial and error.

Relationship Cardinality

The cardinality of the relationship denotes the number of child instances that can be inserted for each parent of that relationship. Cardinality may seem like a fringe topic because the logic to implement can be tricky, but the reality is that if the requirements state the cardinality, it can be important to document the cardinality requirements and to implement restrictions on cardinality in the database constraints where possible. At this

www.it-ebooks.info

point, however, logical modeling is not about how to implement but about documenting what should be. We will talk implementation of data constraints starting in Chapter 6 when we implement our first database, and even more in Chapter 7, the data protection chapter.

Figures 3-1 through 3-6 show the six possible cardinalities that relationships can take. The cardinality indicators are applicable to either mandatory or optional relationships.



Figure 3-1. *One-to-zero or more*



Figure 3-2. *One-to-one or more (at least one), indicated by P*

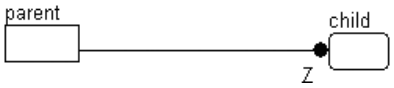


Figure 3-3. *One-to-zero or one (no more than one), indicated by Z*

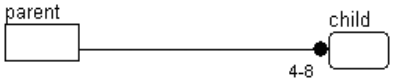


Figure 3-4. *One-to-some fixed range (in this case, between 4 and 8 inclusive)*

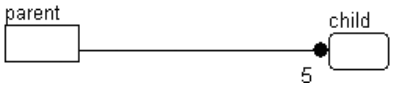


Figure 3-5. *One-to-exactly N (in this case, 5, meaning each parent must have five children)*

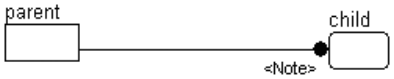
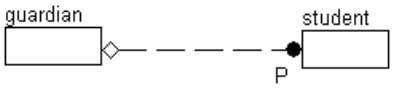


Figure 3-6. *Specialized note describing the cardinality*

For example, a possible use for the one to one-or-more might be to represent the relationship between a guardian and a student in an elementary school database because it would not make sense to have a student at a school without a guardian:

www.it-ebooks.info



This is a good example of a zero-or-one to one-or-more relationship, and a fairly interesting one at that. It says that for a guardian instance to exist, a related student must exist, but a student need not have a guardian for us to wish to store the student's data. Next, let's consider the case of a club that has members with certain positions that they should or could fill, as shown in Figures 3-7 through 3-9.

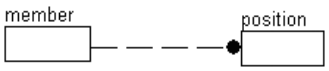


Figure 3-7. *One-to-many allows unlimited positions for the member.*

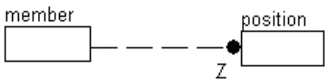


Figure 3-8. *One-to-one allows only one position per member*

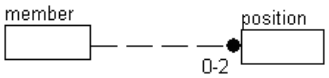


Figure 3-9. *A one-to-zero, one-to-one, or one-to-two relationship specifies a limit of two positions per person.*

Figure 3-7 shows that a member can take as many positions as are possible. Figure 3-8 shows that a member can serve in no position or one position, but no more. Finally, Figure 3-9 shows that a member can serve in zero, one, or two positions. They all look pretty similar, but the Z or 0–2 is important in signifying the cardinality.

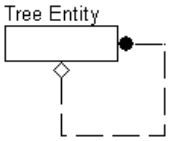
■ **Note** It is a fairly rare occurrence that I have needed anything other than the basic one-many, one-zero, or one relationship types, but your experience may lend itself to the specialized relationship cardinalities.

Recursive Relationships

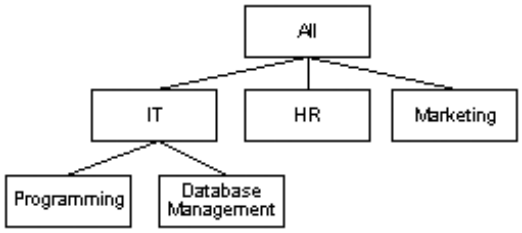
One of the more difficult—and often important—relationship types to implement is the *recursive relationship*, also known as a *self-join*, *hierarchical*, *self-referencing*, or *self-relationship* (I have even heard them referred to as *fish-hook relationships*, but that name always seems silly to me). This is modeled by drawing a nonidentifying

www.it-ebooks.info

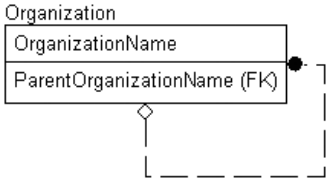
relationship not to a different entity, but to the same entity. The migrated key of the relationship is given a role name. (In many cases, a naming convention of adding "parent" to the front of the attribute name is useful if no natural naming is available.)



The recursive relationship is useful for creating tree structures, as in the following organizational chart:



To explain this concept fully, I will show the data that would be stored to implement this hierarchy:



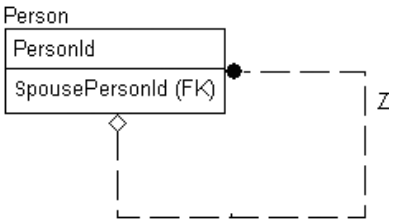
Here is the sample data for this table:

```
OrganizationName      ParentOrganizationName
----------------      ----------------------
All
IT                    ALL
HR                    ALL
Marketing             ALL
Programming           IT
Database Management   IT
```

The organizational chart can now be traversed by starting at `All` and getting the children of `ALL`, for example: `IT`. Then, you get the children of those values, like for `IT` one of the values is `Programming`.

www.it-ebooks.info

As a final example, consider the case of a `Person` entity. If you wanted to associate a person with a single other person as the first person's spouse, you might design the following:

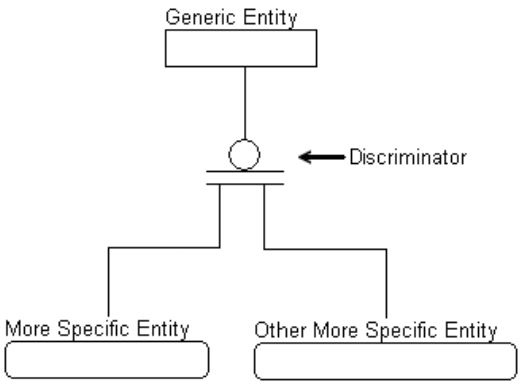


Notice that this is a one-to-zero or one-to-one relationship, since (in most places) a person may have no more than a single spouse but need not have one. If you require one person to be related as a child to two parents, another table entity is required to link two people together.

■ **Note** Hierarchies will be covered in some detail in Chapter 8 when we discuss various modeling patterns and techniques In this chapter, it is just important to understand what the hierarchical relationship looks like on a data model.

Subtypes

Subtypes (also referred to as *categorization relationships*) are another special type of one-to-zero or one-to-one relationship used to indicate whether one entity is a specific type of a generic entity. Note also that there are no black dots at either end of the lines; the specific entities are drawn with rounded corners, signifying that they are, indeed, dependent on the generic entity.

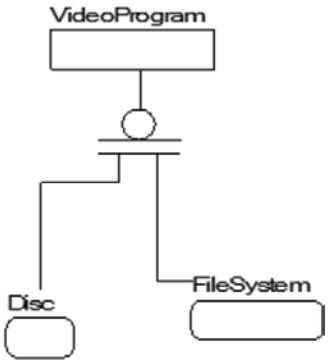


www.it-ebooks.info

There are three distinct parts of the subtype relationship:

- *Generic entity*: This entity contains all of the attributes common to all of the subtyped entities.
- *Discriminator*: This attribute acts as a switch to determine the entity where the additional, more specific information is stored.
- *Specific entity*: This is the place where the specific information is stored, based on the discriminator.

For example, consider an inventory of your home video library. If you wanted to store information about each of the videos that you owned, regardless of format, you might build a categorization relationship like the following:



In this manner, you might represent each video's price, title, actors, length, and possibly description of the content in the `VideoProgram` entity, and then, based on format—which is the discriminator—you might store the information that is specific to `Discs` or `FileSystem` in its own separate entity (e.g., physical location, special features, format [BluRay, DVD, digital copy] for `Disc` based video and directory and format for `FileSystem`).

There are two distinct category types: complete and incomplete. The *complete* set of categories is modeled with a double line on the discriminator, and the *incomplete* set is modeled with a single line (see Figure 3-10).

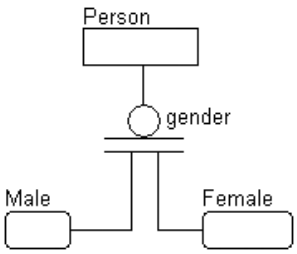


Figure 3-10. *Complete (left) and incomplete (right) sets of categories*

The primary difference between the complete and incomplete categories is that in the complete categorization relationship, each generic instance must have one specific instance, whereas in the incomplete case, this is not necessarily true. An instance of the generic entity can be associated with an instance of only one of the category entities in the cluster, and each instance of a category entity is associated with exactly one instance of the generic entity. In other words, overlapping subentities are not allowed.

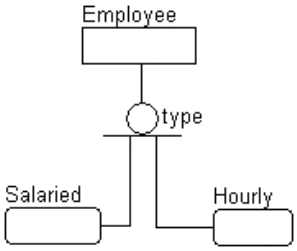
www.it-ebooks.info

For example, you might have a complete set of categories like this:



This relationship is read as follows: "A `Person` *must* be either `Male` or `Female`." This is certainly a complete category. This is not to say that you know the gender of every person in every instance of all entities. Rather, it simply means that if the instance has been categorized, any person must fall in one of the two buckets (male or female).

However, consider the following incomplete set of categories:



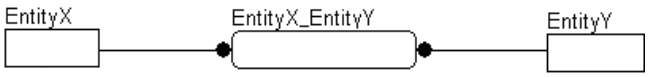
This is an incomplete subtype, because employees are either salaried or hourly, but there may be other categories, such as contract workers. You may not need to store any additional information about them, though, so there is no need to implement the specific entity. This relationship is read as follows: "An `Employee` can be either `Salaried` or `Hourly` or other."

Many-to-Many Relationship

The many-to-many relationship is also known as the *nonspecific relationship*, which is actually a better name, but far less well known. Having quite a few many-to-many relationships in the data model is common, particularly in the early conceptual model. These relationships are modeled by a line with a solid black dot at either end:

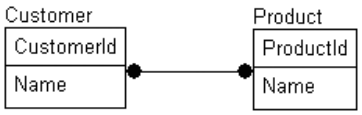


There is one real problem with modeling a many-to-many relationship: it is often necessary to have more information about the relationship than that simply many `EntityX` instances are connected to many `EntityY` instances. So the relationship is usually modeled as follows:



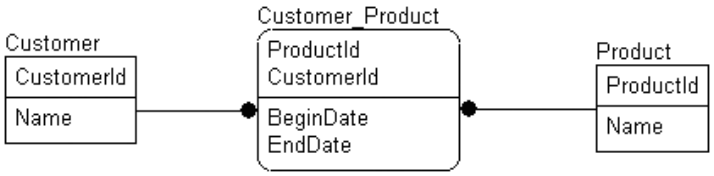
www.it-ebooks.info

Here, the intermediate `EntityX_EntityY` entity is known as an *associative entity* (names like *bridge*, *tweener*, and *joiner* are not uncommon either). In early modeling, I will often stick with the former representation when I haven't identified any extended attributes to describe the relationship and the latter representation when I need to add additional information to the model. To clarify the concept, let's look at the following example:



Here, I have set up a relationship where many customers are related to many products. This situation is common situation because in most cases, companies don't create specific products for specific customers; rather, any customer can purchase any of the company's products. At this point in the modeling, it is reasonable to use the many-to-many representation. Note that I am generalizing the customer-to-product relationship. It is not uncommon to have a company build specific products for only one customer to purchase.

Consider, however, the case where the `Customer` need only be related to a `Product` for a certain period of time. To implement this, you can use the following representation:

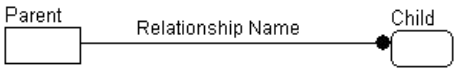


In fact, almost all of the many-to-many relationships tend to require some additional information like this to make them complete. It is not uncommon to have no many-to-many relationships modeled with the black circle on both ends of a model, so you will need to look for entities modeled like this to be able to discern them.

■ **Note** I should also note that you can't implement a many-to-many relationship in SQL without using a table for the resolution. This is because there is no way to migrate keys both ways. In the database, you are required to implement all many-to-many relationships using a resolution entity.

Verb Phrases (Relationship Names)

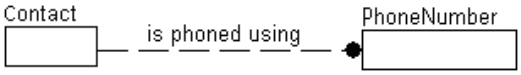
Relationships are given names, called *verb phrases*, to make the relationship between a parent and child entity a readable sentence and to incorporate the entity names and the relationship cardinality. The name is usually expressed from parent to child, but it can be expressed in the other direction, or even in both directions. The verb phrase is located on the model somewhere close to the line that forms the relationship:



The relationship should be named such that it fits into the following general structure for reading the entire relationship: parent cardinality – parent entity name – relationship name – child cardinality – child entity name.

www.it-ebooks.info

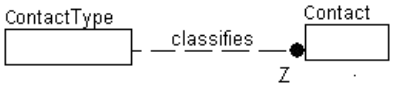
For example, the following relationship



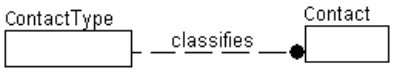
would be read as "one `Contact` is phoned using zero, one, or more `PhoneNumbers`."

Of course, the sentence may or may not make perfect sense in normal conversational language; for example, this one brings up the question of how a contact is phoned using zero phone numbers. If presenting this phrase to a nontechnical person, it would make more sense to read it as follows: "Each contact can have either no phone number or one or more phone numbers."

The modeling language does not take linguistics into consideration when building this specification, but from a technical standpoint, it does not matter that the contact is phoned using zero phone numbers, since it follows that the contact would have no phone number. Being able to read the relationship helps you to notice obvious problems. For instance, consider the following relationship:



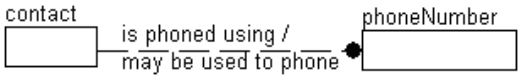
It looks fine at first glance, but when read as "one `contactType` classifies zero or one `Contacts`," it doesn't make logical sense. It means to categorize all of the contacts, a unique `ContactType` row would be required for each `Contact`, which clearly is not at all desirable. This would be properly modeled as follows:



which now reads, "one `contactType` classifies zero or more `Contacts`."

Note that the type of relationship, whether it is identifying, nonidentifying, optional, or mandatory, makes no difference when reading the relationship. You can also include a verb phrase that reads from child to parent. For a one-to-many relationship, this would be of the following format: "One child instance (relationship) exactly one parent instance."

In the case of the first example, you could have added an additional verb phrase:

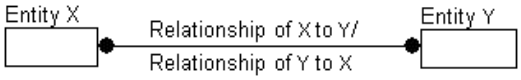


The parent-to-child relationship again is read as "one `Contact` is phoned using zero, one, or more `phoneNumbers`." You can then read the relationship from child to parent. Note that, when reading in this direction, you are in the context of zero or one phone number to one and only one contact: "zero or one `phoneNumbers` may be used to phone exactly one contact."

Since this relationship is going from many to one, the parent in the relationship is assumed to have one related value, and since you are reading in the context of the existence of the child, you can also assume that there is zero or one child record to consider in the sentence.

www.it-ebooks.info

For the many-to-many relationship, the scheme is pretty much the same. As both entities are parents in this kind of relationship, you read the verb phrase written above the line from left to right and from right to left for the verb phrase written below it.



■ **Note** Taking the time to define verb phrases can be a hard sell at times, because they are not actually used in a substantive way in the implementation of the database, and often people consider doing work that doesn't produce code directly to be a waste of time. However, well-defined verb phrases make for great documentation, giving the reader a good idea of why the relationship exists and what it means. I usually use the verb phrase when naming the foreign key constraints too, which you will see in Chapter 6 when we actually create a database with foreign keys.

Descriptive Information

Take a picture of a beautiful mountain, and it will inspire thousands of words about the beauty of the trees, the plants, the babbling brook (my relative ability to describe a landscape being one of the reasons I write technical books). What it *won't* tell you is how to get there yourself, what the temperature is, and whether you should bring a sweater and mittens or your swim trunks.

Data models are the same way. You can get a great start on understanding the database from the model, as I have discussed in the previous sections of this chapter. We started the documentation process by giving good names to entities, attributes, and the relationships, but even with well-formed names, there will still likely be confusion as to what exactly an attribute is used for and how it might be used.

For this, we need to add our own thousand words (give or take) to the pictures in the model. When sharing the model, descriptions will let the eventual reader—and even a future version of yourself—know what you originally had in mind. Remember that not everyone who views the models will be on the same technical level: some will be nonrelational programmers, or indeed users or (nontechnical) product managers who have no modeling experience.

Descriptive information need not be in any special format. It simply needs to be detailed, up to date, and capable of answering as many questions as can be anticipated. Each bit of descriptive information should be stored in a manner that makes it easy for users to quickly connect it to the part of the model where it was used, and it should be stored either in a document or as metadata in a modeling tool.

You should start creating this descriptive text by asking questions such as the following:

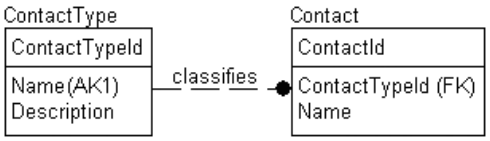
- What is the object supposed to represent?
- How will the object be used?
- Who might use the object?
- What are the future plans for the object?
- What constraints are not specifically implied by the model?

www.it-ebooks.info

The scope of the descriptions should not extend past the object or entities that are affected. For example, the entity description should refer only to the entity, and not any related entities, relationships, or even attributes unless completely necessary. An attribute definition should only speak to the single attribute and where its values might come from.

Maintaining good descriptive information is equivalent to putting decent comments in code. As the eventual database that you are modeling is usually the central part of any computer system, comments at this level are more important than at any others. We can also say that this is the bread and butter of having a logical model. For most people, being able to go back and review notes that were taken about each object and why things were implemented is invaluable, especially true for organizations that bring in new employees and need to bring them up to speed on complex systems.

For example, say the following two entities have been modeled:



The very basic set of descriptive information in Tables 3-1 and 3-2 could be captured to describe the attributes created.

Table 3-1. *Entities*

Entity	Attribute	Description
Contact		Persons that can be contacted to do business with
	ContactId	Surrogate key representing a Contact
	ContactTypeId	Primary key reference for a contactType, classifies the type of contact
	Name	The full name of a contact
ContactType		Domain of different contact types
	ContactTypeId	Surrogate key representing a ContactType
	Name	The name that the contact type will be uniquely known as
	Description	The description of exactly how the contact should be used as

Table 3-2. *Relationships*

Parent Entity Name	Phrase	Child Entity Name	Definition
ContactType	Classifies	Contact	Contact type classification

Alternative Modeling Methodologies

In this section, I will briefly describe a few of the other modeling methodologies that you will likely run into with tools you may use when looking for database information on the Web. You will see a lot of similarities among them—for example, most every methodology uses a rectangle to represent a table and a line to indicate a relationship. You will also see some big differences among them, such as how the cardinality and direction of a relationship is indicated. Where IDEF1X uses a filled circle on the child end and an optional diamond on the

www.it-ebooks.info

other, one of the most popular methodologies uses multiple lines on one end and several dashes to indicate the same things. Still others use an arrow to point from the child to the parent to indicate where the migrated key comes from (that one really confuses people who are used to IDEF1X and crow's feet.)

While all of the examples in this book will be done in IDEF1X, knowing about the other methodologies will be helpful when you are surfing around the Internet, looking for sample diagrams to help you design the database you are working on. (Architects are often particularly bad about not looking for existing designs, because frankly, solving the problem at hand is one of the best parts of the job. However, don't reinvent the wheel every time!)

I will briefly discuss the following:

- *Information engineering (IE)*: The other main methodology, which is commonly referred to as the *crow's feet* method
- *Chen Entity Relationship Model (ERD)*: The methodology used mostly by academics, though you can run into these models online
- *Visio*: A tool that many developers have handy that will do an admirable job of helping you to design a database
- *Management Studio database diagrams*: The database object viewer that can be used to view the database as a diagram right in Management Studio

---

■ **Note** This list is by no means exhaustive. For example, several variations loosely based on the Unified Modeling Language (UML) class modeling methodology are not listed. These types of diagrams are common, particularly with people who use the other components of UML, but these models really have no standards. Some further reading on UML data models can be found in Clare Churcher's book *Beginning Database Design* (Apress, 2007), on Scott Adler's AgileData site (http://www.agiledata.org/essays/umlDataModelingProfile.html), and on IBM's Rational UML documentation site (http://www.ibm.com/software/rational/support/documentation/), and many others. (The typical caveat that these URLs are apt to change applies.)

Information Engineering

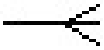
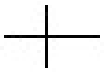
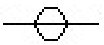
The *information engineering (IE)* methodology is well known and widely used. Like IDEF1X, it does a very good job of displaying the necessary information in a clean, compact manner that is easy to follow. The biggest difference is in how this method denotes relationship cardinalities: it uses a crow's foot instead of a dot and lines and dashes instead of diamonds and some letters.

Tables in this method are denoted as rectangles, basically the same as in IDEF1X. According to the IE standard, attributes are not shown on the model, but most models show them the same as in IDEF1X—as a list, although the primary key is usually denoted by underlining the attributes, rather than the position in the table. (I have seen other ways of denoting the primary key, as well as alternate/foreign keys, but they are all very clear.) Where things get very different using IE is when dealing with relationships.

Just like in IDEF1X, IE has a set of symbols that have to be understood to indicate the cardinality and ownership of the data in the relationships. By varying the basic symbols at the end of the line, you can arrive at all of the various possibilities for relationships. Table 3-3 shows the different symbols that can be employed to build relationship representations.

www.it-ebooks.info

Table 3-3. *Information Engineering Symbols*

Symbol	Relationship Type	Description
	Many	The entity on the end with the crow's foot denotes that there can be greater than one value related to the other entity.
	Required	The key of the entity on the other end of the relationship is required to exist. A line on both ends indicates that a child is required for a parent row to exist, just like a "P" on the end of an IDEF1X model.
	Optional	This symbol indicates that there does not have to be a related instance on this end of the relationship for one to exist on the other. It can appear at the parent or the child.
	Nonrequired	A set of dashed lines on one end of the relationship line indicates that the migrated key may be null.

Figures 3-11 through 3-14 show some examples of relationships in IE.



Figure 3-11. *One-to-many: Specifically, one row in Table A may be related to zero, one or more rows in Table B. A related row must exist in Table A for a row to exist in Table B.*



Figure 3-12. *One-to-many: Specifically, one Row in Table A may be related to one or more rows in Table B. A related row must exist in Table A and a row must exist in Table B.*



Figure 3-13. *One-to-one: Specifically, zero or one row in Table A can be related to zero or one row in Table B. A row needn't exist in Table A for a row to exist in Table B (the key value would be optional).*



Figure 3-14. *Many-to-many relationship*

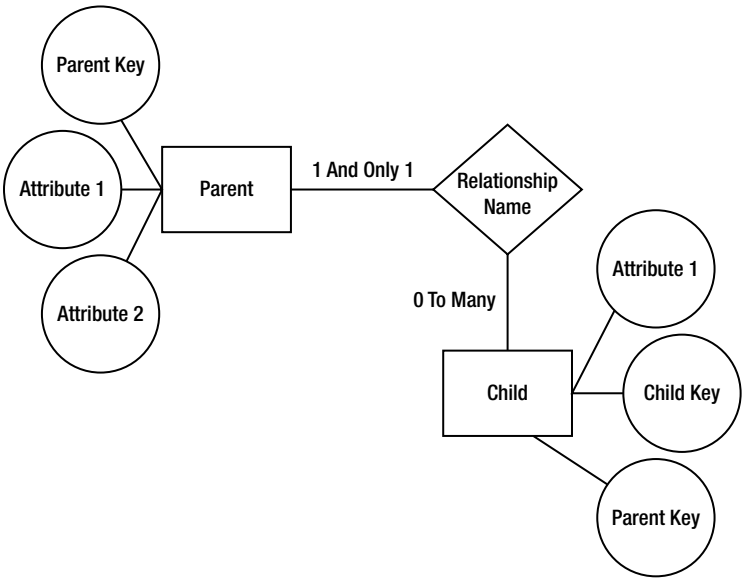
www.it-ebooks.info

I have never felt that this notation was as clean as IDEF1X, because much of the symbology seems a bit complicated to master (I have heard the same about IDEF1X from crow's feet users as well, so it could just be a taste thing). IE conveys the information well though and is likely to be used in some of the documents that you will come across in your work as a data architect or developer. IE is also not always fully implemented in tools; however, usually the circle, dashes, and crow's feet are implemented properly.

■ **Note** You can find more details about the Information Engineering methodology in the book *Information Engineering, Books 1, 2, and 3* by James Martin (Prentice Hall, 1990).

Chen ERD

The Chen Entity Relationship Model (ERD) methodology is quite a bit different from IDEF1X, but it's pretty easy to follow and largely self-explanatory. You will seldom see this methodology anywhere other than in academic texts, but since quite a few of these types of diagrams are on the Internet, it's good to understand the basics of the methodology. Here's a very simple Chen ERD diagram showing the basic constructs:



Each entity is again a rectangle; however, the attributes are not shown in the entity but are instead attached to the entity in circles. The primary key either is not denoted or, in some variations, is underlined. The relationship is denoted by a rhombus, or diamond shape.

The cardinality for a relationship is denoted in text. In the example, it is `1 and Only 1 Parent rows<relationship name>0 to Many Child rows`. The primary reason for including the Chen ERD format is for contrast. Several other modeling methodologies—for example, Object Role Modeling (ORM) and Bachman—implement attributes in this style, where they are not displayed in the rectangle.

www.it-ebooks.info

While I understand the logic behind this approach (entities and attributes are separate things), I have found that models I have seen using the format with attributes attached to the entity like this seemed overly cluttered, even for fairly small diagrams. The methodology does, however, do an admirable job with the logical model of showing what is desired and also does not rely on overcomplicated symbology to describe relationships and cardinality.

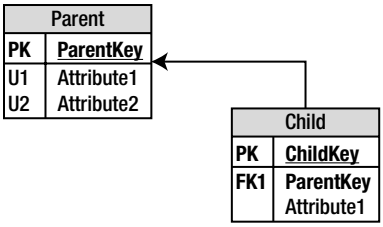
■ **Note** You can find further details on the Chen ERD methodology in the paper "The Entity Relationship Model—Toward a Unified View of Data" by Peter Chen (it can be found by performing a Google search for the title of the paper).

Also, note that I am not saying that such a tool to create Chen diagrams does not exist; rather, I personally have not seen the Chen ERD methodology implemented in a mainstream database design tool other than some early versions of Microsoft Visio. Quite a few of the diagrams you will find on the Internet will be in this style, however, so understanding at least the basics of the Chen ERD methodology is useful.

Visio

Visio is a tool that many developers use for designing databases; often, they already have it in their tool belts for other types of drawings and models (such as process flow diagrams). By nature, Visio is a multipurpose drawing tool and, as such, does not lend itself well to being a fully featured database design tool. That said, Visio is not the world's worst tool to design a database either. It does lack a refined manner of going from conceptual to logical and finally to an implemented database, but unless you are doing serious enterprise-level designs, this limitation may not matter much to you. (Also, like many of us, you may not have the ducats to shell out for a more fully featured tool, and using Visio is better than just using Management Studio's diagrams.)

Models created using Visio have a distinctive style that shares some features with object-oriented design tools. While the tool supports many of the features of more powerful tools, the picture of the model is pretty basic:



The arrow points to the parent in all cases but does not indicate ownership, cardinality, or even optionality. It tells you what columns are primary keys in two ways (using the line and the "PK"), as well as telling you what columns are part of foreign keys with "FK" plus a number, in case there are more than one. Alternate keys are denoted with "U" plus a number. In the preceding model, the `Parent` entity/table has two alternate keys.

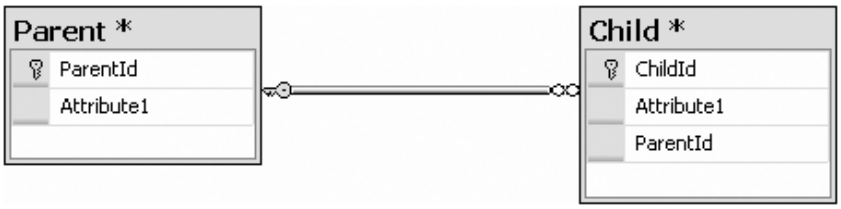
Visio implements a good amount of detail to define columns, include comments, and set cardinalities via dialogs and editors. All in all, the tool is not a terrible choice for modeling if it is the only one available, but far better choices are out there for what, over the lifetime of the tool, will be a pretty reasonable amount of money.

www.it-ebooks.info

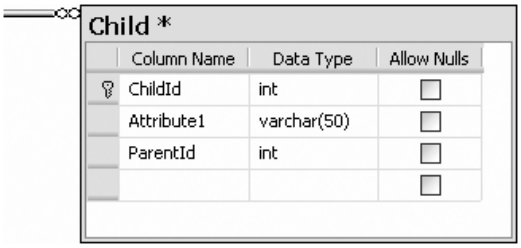
Management Studio Database Diagrams

The database diagramming capability built into SQL Server Management Studio is not a modeling tool, though often, it can be a developer's only available tool for modeling. It is a very useful tool to provide a graphical view the structure of an implemented database (a picture really *is* worth a thousand words!), but because it works directly against the implemented tables, it is not overly useful for design but only for the final implementation. You can use this view to modify structures as well, but I would not suggest it. I typically suggest that one use the T-SQL code in the Management Studio query window to make table changes. (All table structures in this book will be done in this way, and doing so is a best practice for repeatability purposes. I will talk more on this in the latter half of the book.) New to SQL Server 2012 is a tool called SQL Server Developer Tools that will allow you to create a database as well. In this book, I will largely be tool agnostic, choosing to do almost everything via scripts, because it will behoove you as a designer/coder to know what the tools do behind the scenes anyhow.

You will find that Management Studio diagrams are excellent tools when you are looking at a database in an environment where you have no other tools, just to look at the structures. As such, the following is an example of a one-to-many relationship in Management Studio:



The primary keys are identified by the little key in an attribute. The relationship is denoted by the line between the entities, with the "one" end having a key and the "many" end having an infinity sign. You can display the entities in several formats by just showing the names of the entities or by showing all of the attributes with datatypes, for example:



While the database diagram tool does have its place, I must stress that it isn't a full-featured data modeling tool and shouldn't be used as such if you can avoid it. I included coverage of the SQL Server modeling capabilities here because it's included in SQL Server, and in some situations, it's the best tool you may have access to. It does give access to all implementation-specific features of SQL Server, including the ability to annotate your tables and columns with descriptive information. Unfortunately, if you decide to implement a relationship in a trigger, it will not know that the trigger exists. (I cover triggers in Chapter 6 and Chapter 7, so if you have no idea what a trigger is right now, don't worry.)

In most cases, the SQL Server tool isn't the optimal way to see actual relationship information that is designed into the database, but it does offer a serviceable look at the database structure when needed.

www.it-ebooks.info

■ **Note** In the Business Intelligence tools for SQL Server 2005 and later, there is also another tool that resembles a data model in the Data Source view. It is used to build a view of a set of tables, views, and (not implemented) queries for building reports from. It is pretty much self-explanatory as a tool, but it uses an arrow on the parent end of the relation line to indicate where a primary key came from, much like Visio does. This tool is not pertinent to the task of building or changing a database, but I felt I should at least mention it briefly, as it does look very much like a data modeling tool.

Best Practices

The following are some basic best practices that can be very useful to follow when doing data modeling:

- *Modeling language:* Pick a model language, understand it, and use it fully. This chapter has been a basic coverage of much of the symbology of the IDEF1X modeling language. IDEF1X is not the only modeling language, and after using one style, it is likely you will develop a favorite flavor and not like the others (guess which one I like best). The plain fact is that almost all of the modeling options have some merit. The important thing is that you understand your chosen language and can use it to communicate with users and programmers at the levels they need and can explain the intricacies as necessary.
- *Entity names*: There are two ways you can go with these: plural or singular. I feel that names should be singular (meaning that the name of the table describes a single instance, or row, of the entity, much like an OO object name describes the instance of an object, not a group of them), but many other highly regarded data architects and authors feel that the table name refers to the set of rows and should be plural. Whichever way you decide to go, it's most important that you are consistent. Anyone reading your model shouldn't have to guess why some entity names are plural and others aren't.
- *Attribute names*: It's generally not necessary to repeat the entity name in the attribute name, except for the primary key and some common terms. The entity name is implied by the attribute's inclusion in the entity. The attribute name should reflect precisely what is contained in the attribute and how it relates to the entity. And as with entities, abbreviations ought to be used extremely sparingly in naming of attributes and columns; every word should be spelled out in its entirety. If any abbreviation is to be used, because of some naming standard currently in place for example, a method should be put into place to make sure the abbreviation is used consistently.
- *Relationships*: Name relationships with verb phrases, which make the relationship between a parent and child entity a readable sentence. The sentence expresses the relationship using the entity names and the relationship cardinality. The relationship sentence is a very powerful tool for communicating the purpose of the relationships with nontechnical members of the project team (e.g., customer representatives).
- *Domains*: Using defined, reusable domains gives you a set of standard templates to apply when building databases to ensure consistency across your database and, if the templates are used extensively, all of your databases. Implement type inheritance wherever possible to take advantage of domains that are similar and maximize reusability.

www.it-ebooks.info

Summary

One of the primary tools of a database designer is the data model. It's such a great tool because it can show not only the details of single table at a time but the relationships between several entities at a time. Of course, it is not the only way to document a database; each of the following is useful, but not nearly as useful as a full-featured data model:

- Often a product that features a database as the central focus will include a document that lists all tables, datatypes, and relationships.
- Every good DBA has a script of the database saved somewhere for re-creating the database.
- SQL Server's metadata includes ways to add properties to the database to describe the objects.

A good data modeling tool (often a costly thing to purchase but definitely well worth the investment over time) will do all of these things and more for you. I won't give you any guidance as to which tool to purchase, as this is not an advertisement for any tool (not even for the basic Microsoft tools that you likely have your hands on already, which frankly are *not* the best-in-class tools that you need to get). Clearly, you need to do a bit of research to find a tool that suits you.

Of course, you don't need a model to design a database, and I know several very talented data architects who don't use any kind of tools to model with, sticking with SQL scripts to create their databases, so using a modeling tool is not necessary to get the database created. However, a graphical representation of the data model is a very useful tool to quickly share the structure of the database with developers and even end users. And the key to this task is to have common symbology to communicate on a level that all people involved can understand on some level.

In this chapter, I presented the basic process of graphically documenting the objects that were introduced in the first chapter. I focused heavily on the IDEF1X modeling methodology, taking a detailed look at the symbology that will be used through database designs. The base set of symbols outlined here will enable us to fully model logical databases (and later physical databases) in great detail.

All it takes is a little bit of training, and the rest is easy. For example, take the model in Figure 3-15.

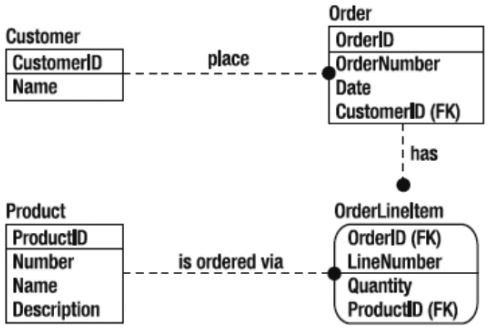


Figure 3-15. *Reading this basic model is not difficult at all, if you simply apply the explanations from this chapter.*

Customers place orders. Orders have line items. The line items are used to order products. With very little effort, nontechnical users can understand your data models, allowing you to communicate very easily, rather than using large spreadsheets as your primary communication method. The finer points of cardinality and ownership might not be completely clear, but usually, those technical details are not as important as the larger picture of knowing which entities relate to which.

www.it-ebooks.info

If you have named attributes well, users won't be confused about what most attributes are, but if so, your spreadsheet of information should be able clear up confusion about the finer points of the model.

Now that we've considered the symbology required to model a database, I'll use data models throughout this book to describe the entities in the conceptual model in Chapter 4, and then, in many other models throughout the implementations presented in the rest of the book as shorthand to give you an overview of the scenario I am setting up, often in addition to scripts to demonstrate how to create the tables in the model.

www.it-ebooks.info

CHAPTER 4

Initial Data Model Production

Good designers can create normalcy out of chaos; they can clearly communicate ideas through the organizing and manipulating of words and pictures.

—Jeffery Veen

In this chapter, we are going to start to apply the skills that were covered in the previous chapters and start creating a data model. It won't be the final model that gets implemented by any means, but the goal of this model will be to serve as the basis for the eventual model that will get implemented.

In some projects, the process of requirements gathering is complete before you start the conceptual data model. Someone has interviewed all the relevant clients (and documented the interviews) and gathered artifacts ranging from previous system documentation to sketches of what the new system might look like to prototypes to whatever is available. In other projects, you may have to model to keep up with your agile team members, and much of the process may get done mentally to produce the part of the model that the programmers are itching to get started with. In either case, the fun part starts now: sifting through all these artifacts and documents (and sometimes dealing with human beings directly) and discovering the database from within this cacophony.

Note In reality, the process of discovery is rarely ever over. Analysts would require a gallon of Jack Bauer's best truth serum to get all of the necessary information from most business users, and the budget of most projects couldn't handle that. In this chapter, I am going to assume the requirements are perfect for simplicity's sake.

The ultimate purpose of the data model is to document and assist in implementing the user community's needs for data in a very structured manner. The conceptual model is the somewhat less structured younger sibling of the final model and will be refined until it produces a logical model and eventually a relational database, using techniques and patterns that we will cover in the next several chapters. The goal, for now, is to simply take the requirements and distill out the stuff that has a purpose in the database. In the rest of this chapter, I'll introduce the following processes:

- *Identifying entities*: Looking for all the concepts that need to be modeled in the database.
- *Identifying relationships between entities*: Relationships between entities are what make entities useful. Here, the goal is to look for natural relationships between high-level entities.

www.it-ebooks.info

- *Identifying attributes and domains*: Looking for the individual data points that describe the entities and how to constrain them to only real/useful values.
- *Identifying business rules*: Looking for the boundaries that are applied to the data in the system that go beyond the domains of a single attribute.
- *Identifying fundamental processes*: Looking for different processes (code and programs) that the client tends to execute that are fundamental to its business.

The result from the first two steps listed is commonly called the *conceptual model*. The conceptual model describes the overall structure of the data model you will be creating so you can checkpoint the process before you get too deep. You will use the conceptual model as a communication device because it has enough structure to show the customer but not so much that a great investment has been made. At this point, you will use the conceptual model as the basis of the logical model by filling in attributes and keys, discovering business rules, and making structural changes to arrive at a picture of the data needs of the client. This early version of the logical model will then go through refinement by following the principles of normalization, which will be covered in the next chapter, to produce a complete logical model that is ready to be translated to a physical data model and implemented as a set of tables, columns, constraints, triggers, and all of the fun stuff that you probably bought this book to read about.

In this chapter, we will go through the steps required to produce an unrefined, early logical model, using a one-page set of requirements as the basis of our design that will be introduced in the first section. For those readers who are new to database design, this deliberate method of working though the design to build this model is a great way to help you follow the best possible process. Take care that I said "new to database design," not "new to creating and manipulating tables in SQL Server." Although these two things are interrelated, they are distinct and different steps of the same process.

After some experience, you will likely never take the time to produce a model exactly like I will discuss in this chapter. In all likelihood, you will perform a lot of these steps mentally and will combine them with some of the refinement processes we will work on in the later chapters. Such an approach is natural and actually a very normal thing. You should know, however, that working though the database design process is a lot like working a complex math problem, in that you are solving a big problem trying to find the answer and showing your work is never a bad thing. As a student in a lot of math classes, I was always amazed that showing your work is usually done more by the advanced mathematician than by anyone else. Advanced people know that writing things down avoids errors, and when errors occur, you can look back and figure out why. This isn't to say that you will never want to go directly from requirements to an implementation model. However, the more you know about how a proper database should look, then the more likely you are to try to force the next model into a certain mold, sometimes without listening to what the customer needs first.

The fact is, building a data model requires a lot of discipline because of our deep-down desire is to just "build stuff." I know I didn't start writing SQL code with a great desire to write and read loads of mind-numbing documentation about software that doesn't yet exist. But tearing off designing structures and writing code without a firm understanding of the requirements leads to pitiful results due to missing important insight into the client's structures and needs, leading you to restructure your solution at least once and possibly multiple times.

Example Scenario

Throughout the rest of the chapter, the following example piece of documentation will be used as the basis of our examples. In a real system, this might be just a single piece of documentation that has been gathered. (It always amazes me how much useful information you can get from a few paragraphs, though to be fair I did write—and rewrite—this example more than a couple of times.)

www.it-ebooks.info

The client manages a couple of dental offices. One is called the Chelsea Office, the other the Downtown Office. The client needs the system to manage its patients and appointments, alerting the patients when their appointments occur, either by e-mail or by phone, and then assisting in the selection of new appointments. The client wants to be able to keep up with the records of all the patients' appointments without having to maintain lots of files. The dentists might spend time at each of the offices throughout the week.

For each appointment, the client needs to have everything documented that went on and then invoice the patient's insurance, if he or she has insurance (otherwise the patient pays). Invoices should be sent within one week after the appointment. Each patient should be able to be associated with other patients in a family for insurance and appointment purposes. We will need to have an address, a phone number (home, mobile, and/or office), and optionally an e-mail address associated with each family, and possibly each patient if the client desires. Currently, the client uses a patient number in its computer system that corresponds to a particular folder that has the patient's records.

The system needs to track and manage several dentists and quite a few dental hygienists who the client needs to allocate to each appointment as well. The client also wants to keep up with its supplies, such as sample toothpastes, toothbrushes, and floss, as well as dental supplies. The client has had problems in the past keeping up with when it's about to run out of supplies and wants this system to take care of this for both locations. For the dental supplies, we need to track usage by employee, especially any changes made in the database to patient records.

Through each of the following sections, our goal will be to acquire all the pieces of information that need to be stored in our new database system. Sounds simple enough, eh? Well, although it's much easier than it might seem, it takes time and effort (two things every programmer has in abundance, right?).

The exercise/process provided in this chapter will be similar to what you may go through with a real system design effort, but it is very much incomplete. The point of this chapter is to give you a feeling for how to extract a data model from requirements. The requirements in this section are very much a subset of what is needed to implement the full system that this dental office will need. In the coming chapters, we will present smaller examples to demonstrate independent concepts in modeling that have been trimmed down to only the concepts needed.

Identifying Entities

Entities generally represent people, places, objects, ideas, or things referred to grammatically as nouns. While it isn't really critical for the final design to put every noun into a specific bucket of types, it can be useful in identifying patterns of attributes later. People usually have names, phone numbers, and so on. Places have an address that identifies an actual location.

It isn't critical to identify that an entity is a person, place, object, or idea, and in the final database, it won't make a bit of difference. However, in the next major section of this chapter, we will use these types as clues to some attribute needs and to keep you on the lookout for additional bits of information along the way. So I try to make a habit of classifying entities as people, places, and objects for later in the process. For example, our dental office includes the following:

- People: A patient, a doctor, a hygienist, and so on
- Place: Dental office, patient's home, hospital
- Object: A dental tool, stickers for the kids, toothpaste

www.it-ebooks.info

- Idea: A document, insurance, a group (such as a security group for an application), the list of services provided, and so on

There's clearly overlap in several of the categories (for example, a building is a "place" or an "object"). Don't be surprised if some objects fit into several of the subcategories below them that I will introduce. Let's look at each of these types of entities and see what kinds of things can be discovered from the documentation sample in each of the aforementioned entity types.

■ **Tip** The way an entity is implemented in a table might be different from your initial expectation. It's better not to worry about such details at this stage in the design process—you should try hard not to get too wrapped up in the eventual database implementation. When building the initial design, you want the document to come initially from what the user wants. Then, you'll fit what the user wants into a proper table design later in the process. Especially during the conceptual modeling phase, a change in the design is a click and a drag away, because all you're doing is specifying the foundation; the rest of the house shouldn't be built yet.

People

Nearly every database needs to store information about people. Most databases have at least some notion of user (generally thought of as people, though not always so don't assume and end up with a first name of "Alarm" and last name "System"). As far as real people are concerned, a database might need to store information about many different types of people. For instance, a school's database might have a `student` entity, a `teacher` entity, and an `administrator` entity.

In our example, four people entities can be found—patients, dentists, hygienists, and employees:

> *. . . the system to manage its* `patients` *. . .*
>
> and
>
> *. . . manage several* `dentists`*, and quite a few dental* `hygienists` *. . .*

Patients are clearly people, as are dentists and hygienists (yes, that crazy person wearing a mask that is digging into your gums with a pitchfork is actually a person). Because they're people, specific attributes can be inferred (such as that they have names, for example).

One additional person type entity is also found here:

> *. . . we need to track usage by* `employee` *. . .*

Dentists and hygienists have already been mentioned. It's clear that they'll be employees as well. For now, unless you can clearly discern that one entity is exactly the same thing as another, just document that there are four entities: patients, hygienists, dentists, and employees. Our model then starts out as shown in Figure 4-1.

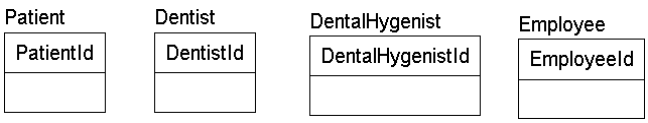


Figure 4-1. *Four entities that make up our initial model*

www.it-ebooks.info

■ **Tip** Note that I have started with giving each entity a simple surrogate key attribute. In the conceptual model, we don't care about the existence of a key, but as we reach the next step with regards to relationships, the surrogate key will migrate from table to table to give a clear picture of the lineage of ownership in the model. Feel free to leave the surrogate key off if you want, especially if it gets in the way of communication, because lay people sometimes get hung up over keys and key structures.

Places

Users will want to store information relative to many different types of places. One obvious place entity is in our sample set of notes:

> *. . . manages a couple of dental offices . . .*

From the fact that dental offices are places, later we'll be able to infer that there's address information about the offices, and probably phone numbers, staffing concerns, and so on. We also get the idea from the requirements that the two offices aren't located very close to each other, so there might be business rules about having appointments at different offices or to prevent the situation in which a dentist might be scheduled at two offices at one time. "Inferring" is just slightly informed guessing, so verify all inferences with the client.

I add the `Office` entity to the model, as shown in Figure 4-2.



Figure 4-2. *Added Office as an entity*

■ **Note** To show progress in the model as it relates to the narrative in the book, in the models, things that haven't changed from the previous step in the process are in gray, while new things are uncolored.

Objects

Objects refer primarily to physical items. In our example, there are a few different objects:

> *. . . with its supplies, such as sample toothpastes, toothbrushes, and floss, as well as dental supplies . . .*

Supplies, such as sample toothpastes, toothbrushes, and floss, as well as dental supplies, are all things that the client needs to run its business. Obviously, most of the supplies will be simple, and the client won't need to store a large amount of descriptive information about them. For example, it's possible to come up with a pretty intense list of things you might know about something as simple as a tube of toothpaste:

- Tube size: Perhaps the length of the tube or the amount in grams
- Brand: Colgate, Crest, or some off-brand

www.it-ebooks.info

- Format: Metal tube, pump, and so on
- Flavor: Mint, bubble gum (the nastiest of all flavors), cinnamon, and orange
- Manufacturer information: Batch number, expiration date, and so on

We could go on and on coming up with more and more attributes of a tube of toothpaste, but it's unlikely that the users will have a business need for this information, because they probably just have a box of whatever they have and give it out to their patients (to make them feel better about the metal against enamel experience they have just gone through). One of the first lessons about over-engineering starts right here. At this point, we need to apply *selective ignorance* to the process and ignore the different attributes of things that have no specifically stated business interest. If you think that the information is useful, it is probably a good idea to drill into the client's process to make sure what they actually want, but don't assume that just because you could design the database to store something that it is necessary, or that the client will change their processes to match your design. If you have good ideas they might, but most companies have what seem like insane business rules for reasons that make sense to them and they can reasonably defend them.

Only one entity is necessary—`Supply`—but document that "Examples given were sample items, such as toothpaste or toothbrushes, plus there was mention of dental supplies. These supplies are the ones that the dentist and hygienists use to perform their job." This documentation you write will be important later when you are wondering what kind of supplies are being referenced.

Catching up the model, I add the `Supply` entity to the model, as shown in Figure 4-3.

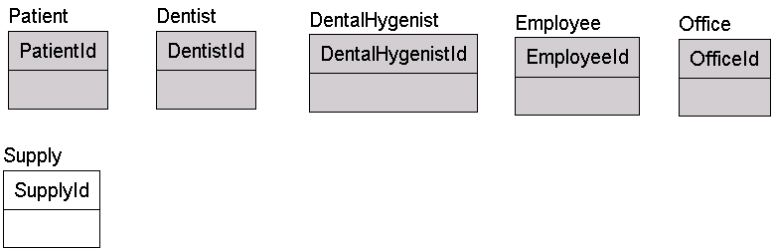


Figure 4-3. *Added the Supply entity*

Ideas

No law or rule requires that entities should represent real objects or even something that might exist physically. At this stage of discovery, you need to consider information on objects that the user wants to store that don't fit the already established "people," "places," and "objects" categories and that might or might not be physical objects.

For example, consider the following:

> *. . . and then invoice the patient's* `insurance`*, if he or she has insurance (otherwise the* `patient pays`*)...*

`Insurance` is an obvious important entity as the medical universe rotates around it. Another entity name looks like a verb rather than a noun in the phrase "patient pays." From this, we can infer that there might be some form of payment entity to deal with.

www.it-ebooks.info

■ **Tip** Not all entities will be adorned with a sign flashing "Yoo-hoo, I am an entity!" A lot of the time, you'll have to read what has been documented over and over and sniff it out like a pig on a truffle.

The model now looks like Figure 4-4.

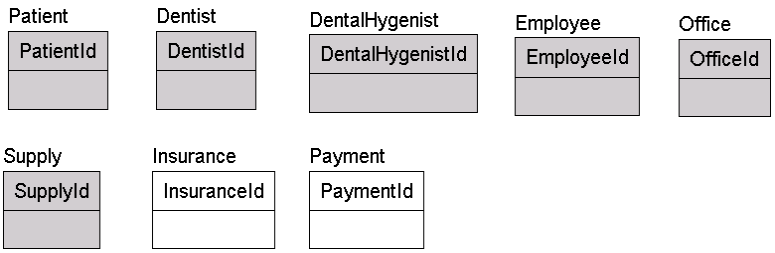


Figure 4-4. *Added the Insurance and Payment entities*

Documents

A document represents some piece of information that is captured and transported in one package. The classic example is a piece of paper that has been written on, documenting a bill that needs to be paid. If you have a computer and/or have used the Interweb at all, you probably know that the notion that a document has to be a physical piece of paper is as antiquated as borrowing a cup of sugar from your neighbor. And even for a paper document, what if someone makes a copy of the piece of paper? Does that mean there are two documents, or are they both the same document? Usually, it isn't the case, but sometimes people do need to track physical pieces of paper and, just as often, versions and revisions of a document.

In the requirements for our new system, we have a few examples of documents that need to be dealt with. First up, we have:

> *. . . and then* `invoice` *the patient's insurance, if he or she has insurance (otherwise the patient pays) . . .*

Invoices are pieces of paper (or e-mails) that are sent to a customer after the services have been rendered. However, no mention was made as to how invoices are delivered. They could be e-mailed or postal mailed—it isn't clear—nor would it be prudent for the database design to force it to be done either way unless this is a specific business rule. At this point, just identify the entities and move along; again, it usually isn't worth it to spend too much time guessing how the data will be used. This is something you should interview the client for.

Next up, we have the following:

> *. . . appointments,* `alerting` *the patients when their appointments occur, either by e-mail or by phone . . .*

This type of document almost certainly isn't delivered by paper but by an e-mail message or phone call. The e-mail is also used as part of another entity, an `Alert`. The alert can be either an e-mail or a phone alert. You may also be thinking "Is the alert really something that is stored?" Maybe or maybe not, but it is probably likely

www.it-ebooks.info

that when the administrative assistants call and alert the patient that they have an appointment, a record of this interaction would be made. Then, when the person misses their appointment, they can say, "We called you!"

■ **Note** If you are alert, you probably are thinking that `Appointment`, `Email`, and `Phone` are all entity possibilities, and you would be right. Here, I am looking at the types individually to make a point. In the real process, you would just look for nouns linearly through the text.

Next, we add the `Invoice` and `Alert` entities to the model, as shown in Figure 4-5.

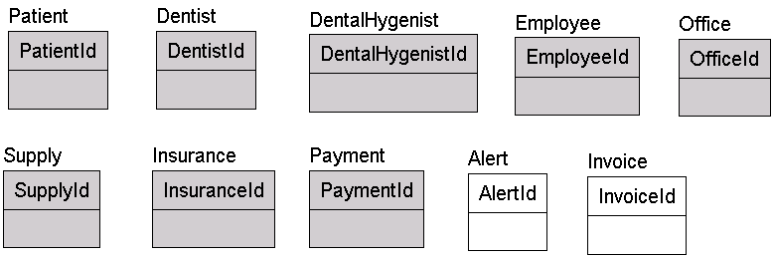


Figure 4-5. *Added the Alert and Invoice entities*

Groups

Another idea-type entity is a group of things, or more technically, a grouping of entities. For example, you might have a club that has members or certain types of products that make up a grouping that seems more than just a simple attribute. In our sample, we have one such entity:

> *Each patient should be able to be associated with other patients in a* `family` *for insurance and appointment purposes.*

Although a person's family is an attribute of the person, it's more than that. So, we add a `Family` entity, as shown in Figure 4-6.

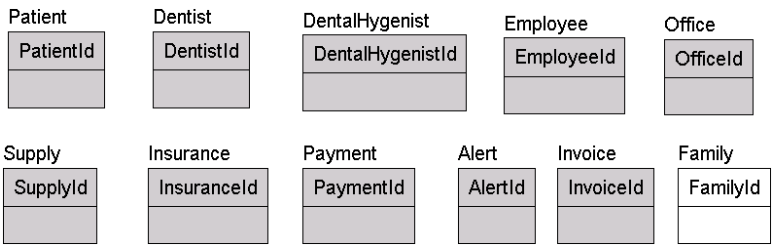


Figure 4-6. *Added the Family entity*

www.it-ebooks.info

Other Entities

The following sections outline some additional common objects that are perhaps not as obvious as the ones that have been presented. They don't always fit a simple categorization, but they're pretty straightforward.

Audit Trails

Audit trails, generally speaking, are used to track changes to the database. You might know that the RDBMS uses a log to track changes, but this is off-limits to the average user. So, in cases where the user wants to keep up with who does what, entities need to be modeled to represent these logs. They could be analogous to a sign-in/sign-out sheet, an old-fashioned library card in the back of the book, or just a list of things that went on in any order.

Consider the following example:

> *For the dental supplies, we need to track usage by employee, and especially any changes made in the database to the patient records.*

In this case, the client clearly is keen to keep up with the kinds of materials that are being used by each of its employees. Perhaps a guess can be made that the user needs to be documented when dental supplies are taken (the difference between dental supplies and non-dental supplies will certainly have to be discussed in due time). Also, it isn't necessary at this time that the needed logging be done totally on a computer, or even by using a computer at all.

A second example of an audit trail is as follows:

> *For the dental supplies, we need to track usage by employee, `and especially any changes made in the database to the patient records`.*

A typical entity that you need to define is the audit trail or a log of database activity, and this entity is especially important when the data is sensitive. An audit trail isn't a normal type of entity, in that it stores no data that the user directly manipulates, and the final design will generally be deferred to the implementation design stage, although it is common to have specific requirements for what sorts of information need to be captured in the audit. Generally, the primary kinds of entities to be concerned with at this point are those that users wish to store in directly.

Events

Event entities generally represent verbs or actions:

> *For each `appointment`, the client needs to have everything documented that went on . . .*

An appointment is an event, in that it's used to record information about when patients come to the office to be tortured for not flossing regularly enough. For most events, appointments included, it's important to have a schedule of when the event is (or was) and where the event will or did occur. It's also not uncommon to want to have data that documents an event's occurrence (what was done, how many people attended, and so on). Hence, many event entities will be tightly related to some form of document entity. In our example, appointments are more than likely scheduled for the future, along with information about the expected activities (cleaning, x-rays, etc.), and when the appointment occurs, a record is made of what services were actually performed so that the dentist can get paid. Generally speaking, there are all sorts of events to look for in any system, such as meter readings for a utility company, weather readings for a sensor, equipment measurements, phone calls, and so on.

www.it-ebooks.info

Records and Journals

The last of the entity types to examine at this stage is a record or journal of activities. Note that I mean "record" in a nondatabase sort of way. A record could be any kind of activity that a user might previously have recorded on paper. In our example, the user wants to keep a record of each visit:

> *The client wants to be able to keep up with the records of all the patients' appointments without having to maintain lots of files.*

Keeping information in a centralized database of the main advantages of building database systems: eliminating paper files and making data more accessible, particularly for future data mining. How many times must I tell the doctor what medicines I'm taking, all because her files are insane clutters used to cover her billing process, rather than being a useful document of my history? What if I forget one that another doctor prescribed, and it interacts strongly with another drug? All of this duplication of asking me what drugs I take is great for covering the doctor's and pharmacy's hides, but by leveraging an electronic database that is connected to other doctor and pharmacy records, the information that people are constantly gathering comes alive, and trends can be seen instantly in ways it would take hours to see on paper. "Hmm, after your primary doctor started you taking Vitamin Q daily, when the time between cleanings is more than 10 months from the previous one, you have gotten a cavity!" Of course, putting too much information in a centralized location makes security all that much more important as well, so it is a double-edged sword as well (we will talk security in Chapter 9, but suffice it to say that it is not a trivial issue).

The model after the changes looks like Figure 4-7.

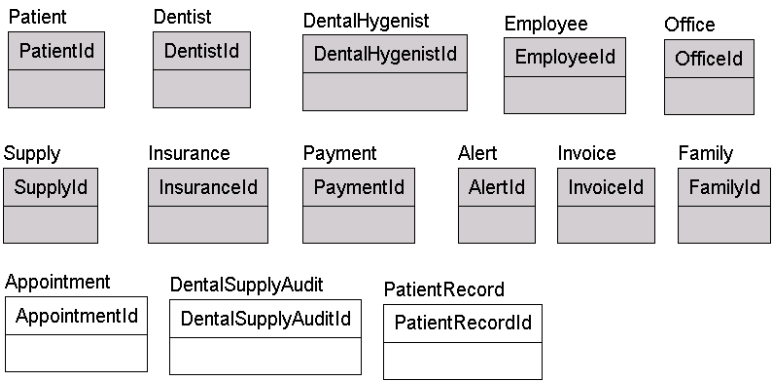


Figure 4-7. *Added the Appointment, DentalSupplyAudit, and PatientRecord entities*

Careful now, as you are probably jumping to a quick conclusion that a payment record is just an amalgamation of their invoices, insurance information, x-rays, and so on. This is 100 % true, and I will admit right now that the `PatientRecord` table is *probably* a screen in the application where each of the related rows would be located. During early conceptual modeling, it is often best to just put it on the model and do the refinement later when you feel you have everything needed to model it right.

Entity Recap

So far, we've discovered the list of preliminary entities shown in Table 4-1. It makes a pretty weak model, but this will change in the next few sections as we begin adding relationships between entities and the attributes. Before progressing any further, stop, define, and document the entities as shown in Table 4-1.

www.it-ebooks.info

Table 4.1. *Entity Listing*

Entity	Type	Description
Patient	People	The people who are the customers of the dental office. Services are performed, supplies are used, and patients are billed for them.
Family	Idea	A group of patients grouped together for convenience.
Dentist	People	People who do the most important work at the dental office. Several dentists are working for the client's practice.
Hygienists	People	People who do the basic work for the dentist. There are quite a few more hygienists than dentists. *(Note: Check with client to see whether there are guidelines for the number of hygienists per dentist. Might be needed for setting appointments.)*
Employee	People	Any person who works at the dental office. Dentists and hygienists are clearly a type of employee.
Office	Places	Locations where the dentists do their business. They have multiple offices to deal with and schedule patients for.
Supplies	Objects	Examples given were sample items, such as toothpaste or toothbrushes, plus there was mention of dental supplies. These supplies are the ones that the dentist and hygienists use to perform their job.
Insurance	Idea	Used by patients to pay for the dental services–rendered work.
Payment	Idea	Money taken from insurance or patients (or both) to pay for services.
Invoice	Document	A document sent to the patient or insurance company explaining how much money is required to pay for services.
Alert	Document	E-mail or phone call made to tell patient of an impending appointment.
Dental Supply Audit	Audit Trail	Used to track the usage of dental supplies.
Appointment	Event	The event of a patient coming in and having some dental work done.
Patient Record	Record	All the pertinent information about a patient, much like a patient's folder in any doctor's office.

Implementation modeling note: log any changes to sensitive/important data.

The descriptions are based on the facts that have been derived from the preliminary documentation. Note that the entities that have been specified are directly represented in the customer's documentation.

Are these all of the entities? Maybe, maybe not, but it is what we have discovered after the first design pass. During a real project you will frequently discover new entities and delete an entity or two that you thought would be necessary. It is a not a perfect process in most cases, because you will be constantly learning the needs of the users.

■ **Note** The conceptual modeling phase is where knowledge of your clients' type of business can help and hinder you. On one hand, it helps you see what they want quickly, but at the same time it can lead you to jump to conclusions based on "how things were done at my other client." Every client is unique and has its own way of doing stuff. The most important tools you will need to use are your ears.

www.it-ebooks.info

Relationships between Entities

Next, we will look for the ways that the entities relate to one another, which will then be translated to relationships between the entities on the model. The idea here is to find how each of the entities will work with one another to solve the client's needs. I'll start first with the one-to-N type of relationships and then cover the many-to-many. It's also important to consider elementary relationships that aren't directly mentioned in your requirements, but it can't be stressed enough that making too many inferences at this point in the process can be detrimental to the process. The user knows what they want the system to be like, and you are going to go back over their requests and fill in the blanks later.

One-to-N Relationships

In each of the one-to-N (commonly one-to-one or one-to-many) relationships, the table that is the "one" table in the relationship is considered the parent, and the "N" is the child or children rows. While the one-to-N relationship is going to be the only relationship you will implement in your relational model, a lot of the natural relationships you will discover in the model may in fact turn out to be many-to-many relationships. It is important to really scrutinize the cardinality of all relationships you model so as not to limit future design considerations by missing something that is very natural to the process. To make it more real, instead of thinking about a mechanical term like One-to-N, we will break it down in to a couple of types of relationships:

- *Associative*: – A parent type is related to one or more child types. The primary point of identifying relationships this way is to form an association between two entity rows.
- *Is a*: Unlike the previous classification, when we think of an is-a relationship, usually the two related items are the same thing, often meaning that one table is a more generic version of the other. A manager is an employee, for example. Usually, you can read the relationship backwards, and it will make sense as well.

I'll present examples of each type in the next couple sections.

■ **Tip** A good deal of the time required to do database design may be simple "thinking" time. I try to spend as much time thinking about the problem as I do documenting/modeling a problem, and often more. It can make your management uneasy (especially if you are working an hourly contract), since "thinking" looks a lot like "napping" (occasionally because they end up being the same thing), but working out all the angles in your head is definitely worth it.

Associative Relationships

In this section, we discuss some of the types of associations that you might uncover along the way as you are modeling relationships. Another common term for an associative relationship is a has-a relationship, so named because as you start to give verb phrase/names to relationships, you will find it very easy to say "has a" for almost every associative type relationship. In fact, a common mistake by modelers is to end up using "has a" as the verb phrase for too many of their parent-child relationships.

In this section, I will discuss a few example type relationships that you will come in contact quite often. The following are different types of has-a relationships:

- *Association*: A simple association between two things, like a person and their driver's license or car. This is the most generic of all relationships.

www.it-ebooks.info

- *Transaction*: Or perhaps more generically, this could be thought of as an interaction relationship. For example, a customer pays bill or makes phone call, so an account is credited/debited money through transactions.
- *Multivalued attribute*: In an object implemented in an object oriented language, one can have arrays for attributes, but in a relational database, as discussed in Chapter 1, all attributes must be atomic/scalar values (or at least should be) in terms of what data we will use in relational queries. So when we design, say, an invoice entity, we don't put all of the line items in the same table, we make a new table, commonly called `invoiceLineItem`. Another common example is storing customers' preferences. Unless they can only have a single preference, a new table is required for each.

In the next few sections, I will use these three types of relationships to classify and pick out some of the relationship types discovered in our dental example.

Association

In our example requirements paragraph, consider the following:

> *... then invoice the patient's insurance, if he or she has insurance...*

In this case, the relationship is between the `Patient` and `Insurance` entities. It's an optional relationship, because it says "if he or she has insurance." Add the following relationship to the model, as shown in Figure 4-8.

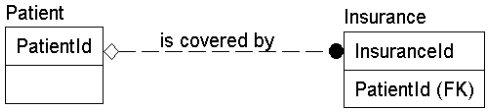


Figure 4-8. *Added the relationship between the Patient and Insurance entities*

Another example of a has-a relationship follows:

> *Each patient should be able to be associated with other patients in a family for insurance and appointment purposes.*

In this case, we identify that a family has patients. Although this sounds a bit odd, it makes perfect sense in the context of a medical office. Instead of maintaining ten different insurance policies for each member of a family of ten, the client wants to have a single one where possible. So, we add a relationship between family and patient, stating that a family instance may have multiple patient instances. Note too that we make it an optional relationship because a patient isn't required to have insurance.

That the family is covered by insurance is also a possible relationship in Figure 4-9. It has already been specified that patients have insurance. This isn't unlikely, because even if a person's family has insurance, one of the members might have an alternative insurance plan. It also doesn't contradict our earlier notion that patients have insurance, although it does give the client two different paths to identify the insurance. This isn't necessarily a problem, but when two insurance policies exist, you might have to implement business rule logic to decide which one takes precedence. Again, this is something to discuss with the client and probably not something to start making up.

www.it-ebooks.info

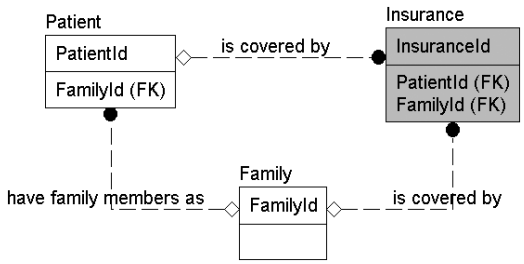


Figure 4-9. *Relationships added among the Patient, Insurance, and Family entities*

Here's another example of a has-a relationship, shown in Figure 4-10:

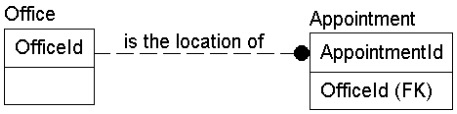


Figure 4-10. *Relationship added between the Office and Appointment entities*

...dental offices . . . The client needs the system to manage its patients and appointments...

In this case, make a note that each dental office will have appointments. Clearly, an appointment can be for only a single dental office, so this is a has-a relationship. One of the attributes of an event type of entity is a location. It's unclear at this point whether a patient comes to only one of the offices or whether the patient can float between offices. This will be a question for the clients when you go back to get clarification on your design. Again, these relationships are optional because a patient is not required to be a member of a family in the client's instructions.

Transactions

Transactions are probably the most common type of relationships in databases. Almost every database will have some way of recording interactions with an entity instance. For example, some very common transaction are simply customers making purchases, payments, and so on. Generally speaking, capturing transactions is how you know that business is good. I can't imagine a useful database that only has customer and product data with transactional information recorded elsewhere.

In our database, we have one very obvious transaction:

...if he or she has insurance (otherwise the patient pays). Invoices should be sent

We identified patient and payment entities earlier, so we add a relationship to represent a patient making a payment. Figure 4-11 shows the new relationship.

www.it-ebooks.info

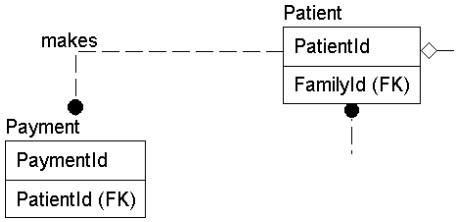


Figure 4-11. *Relationship added between the Patient and Appointment entities*

Multivalued Attributes

During the early phases of modeling, it is far less likely to discover multivalued relationships naturally than any other types. The reason is that users generally think in large concepts, such as objects. In our model, so far, we identified a couple of places where we are likely to expand the entities to cover array types that may not strictly be written in requirements, particularly true when the requirements are written by end users. (`Invoices` have invoice line items; `Appointments` have lists of `Actions` that will be taken such as cleaning, taking x-rays, and drilling; `Payments` can have multiple payment sources.) Therefore, I won't come up with any examples of multivalued attributes in the example paragraphs, but we will cover more about this topic in Chapter 8 when I cover modeling patterns for implementation.

The Is-A Relationship

The major idea behind an is-a relationship is that the child entity in the relationship *extends* the parent. For example, cars, trucks, RVs, and so on, are all types of vehicles, so a car is a vehicle. The cardinality of this relationship is always one-to-one, because the child entity simply contains more specific information that qualifies this extended relationship. There would be some information that's common to each of the child entities (stored as attributes of the parent entity) but also other information that's specific to each child entity (stored as attributes of the child entity).

In our example text, the following snippets exist:

> *. . . manage several* `dentists`*, and quite a few* `dental hygienists` *who the client . . .*
>
> and
>
> *. . . track usage by* `employee`*, and especially . . .*

From these statements, you can reasonably infer that there are three entities, and there's a relationship between them. A dentist is an employee, as is a dental hygienist. There are possibly other employees for whom the system needs to track supply usage as well. Figure 4-12 represents this relationship.

■ **Note** Because the subtype manifests itself as a one-to-one identifying relationship (recall from Chapter 3 that the Z on the relationship line indicates a one-to-one relationship), separate keys for the `Dentist` and `DentalHygienist` entities aren't needed.

This use of keys can be confusing in the implementation, since you might have relationships at any of the three table levels and since the key will have the same name. These kinds of issues are why you maintain a data model for the user to view as needed to understand the relationships between tables.

---

www.it-ebooks.info

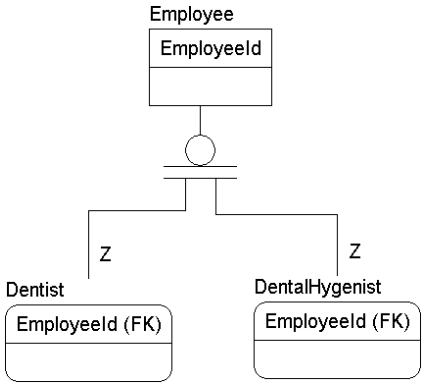


Figure 4-12. *Identified subtyped relationship between the Employee, Dentist, and DentalHygenist entities*

Many-to-Many Relationships

Many-to-many relationships are far more prevalent than you might think. In fact, as you refine the model, a great number of relationships may end up being many-to-many relationships as the real relationship between entities is realized. However, early in the design process, only a few obvious many-to-many relationships might be recognized. In our example, one is obvious:

The dentists might spend time at each of the offices throughout the week.

In this case, multiple dentists can work at more than one dental office. A one-to-many relationship won't suffice; it's wrong to state that one dentist can work at many dental offices, because this implies that each dental office has only one dentist. The opposite, that one office can support many dentists, implies dentists work at only one office. Hence, this is a many-to-many relationship (see Figure 4-13).

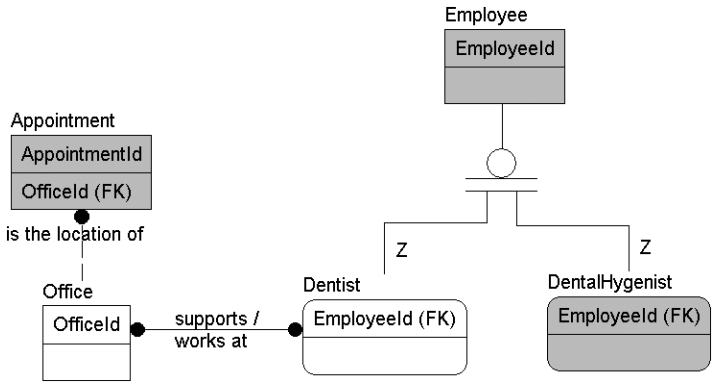


Figure 4-13. *Added a many-to-many relationship between Dentist and Office*

www.it-ebooks.info

I know what most of you are thinking, "Hey what about dentists being associated with appointments, and the same for dental hygienists?" First off, that is good thinking. When you get back to your client, you probably will want to discuss that issue with them. For now, we document what the requirements ask for, and later, we ask the analyst and the client if they want to track that information. It could be that, in this iteration of the product, they just want to know where the dentist is so they can use that information when making manual appointments. Again, we are not to read minds but to do what the client wants in the best way possible.

This is an additional many-to-many relationship that can be identified:

. . . *dental supplies, we need to track usage by employee* . . .

This quote says that multiple employees can use different types of supplies, and for every dental supply, multiple types of employees can use them. However, it's possible that controls might be required to manage the types of dental supplies that each employee might use, especially if some of the supplies are regulated in some way (such as narcotics).

The relationship shown in Figure 4-14 is added.

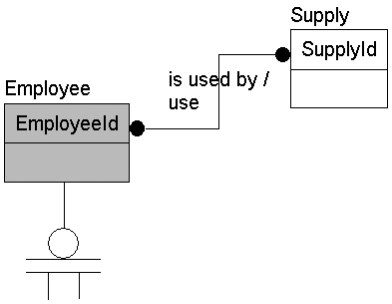


Figure 4-14. *Added a many-to-many relationship between the Supply and Employee entities*

I'm also going to remove the `DentalSupplyAudit` entity, because it's becoming clear that this entity is a report (in a real situation, you'd ask the client to make sure, but in this case, I'm the client, and I agree).

Listing Relationships

Figure 4-15 shows the model so far.

www.it-ebooks.info

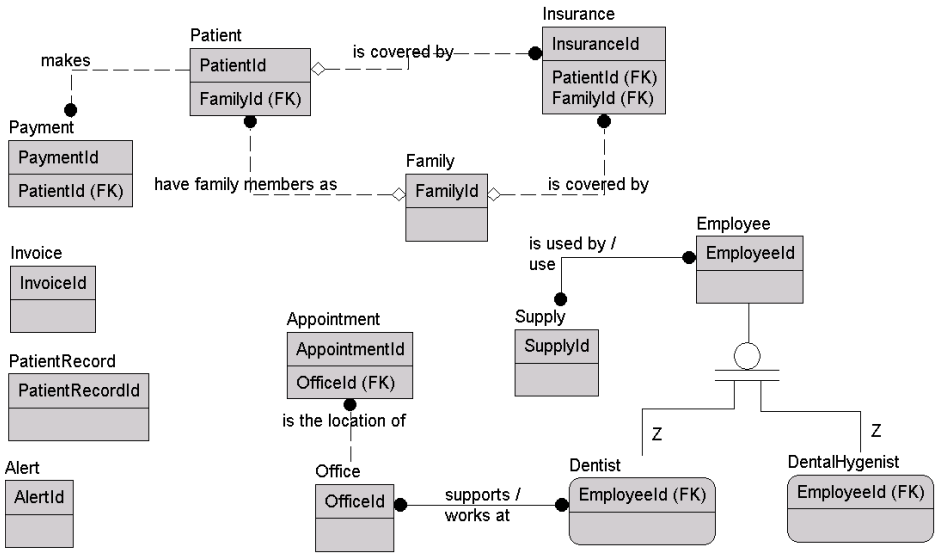


Figure 4-15. *The model so far*

There are other relationships in the text that I won't cover explicitly, but I've documented them in the descriptions in Table 4-2, which is followed by the model with relationships identified and the definitions of the relationships in our documentation (note that the relationship is documented at the parent only).

Table 4.2. *Iniztial Relationship Documentation*

Entity	Type	Description
`Patient`	People	The people who are the customers of the dental office. Services are performed, supplies are used, and the patient is billed for these services.
	Is covered by `Insurance`	Identifies when the patient has personal insurance.
	Is reminded by `Alerts`	Alerts are sent to patients to remind them of their appointments.
	Is scheduled via `Appointments`	Appointments need to have one patient.
	Is billed with `Invoices`	Patients are charged for appointments via an invoice.
	Makes `Payment`	Patients make payments for invoices they receive.
	Has activity listed in `PatientRecord`	Activities that happen in the doctor's office.
`Family`	Idea	A group of patients grouped together for convenience.
	Has family members as `Patients`	A family consists of multiple patients.
	Is covered by `Insurance`	Identifies when there's coverage for the entire family.

(continued)

www.it-ebooks.info

Table 4.2. *(continued)*

Entity	Type	Description
Dentist	People	People who do the most important work at the dental office. Several dentists work for the client's practice.
	Works at many Offices	Dentists can work at many offices.
	Is an Employee	Dentists have some of the attributes of all employees.
	Works during Appointments	Appointments might require the services of one dentist.
Hygienist	People	People who do the basic work for the dentist. There are quite a few more hygienists than dentists. *(Note: Check with client to see if there are guidelines for the number of hygienists per dentist. Might be needed for setting appointments.)*
	Is an Employee	Hygienists have some of the attributes of all employees.
	Has Appointments	All appointments need to have at least one hygienist.
Employee	People	Any person who works at the dental office. Dentists and hygienists are clearly types of employees.
	Use Supplies	Employees use supplies for various reasons.
Office	Places	Locations where the dentists do their business. They have multiple offices to deal with and schedule patients for.
	Is the location of Appointments	Appointments are made for a single office.
Supplies	Objects	Examples given were sample items, such as toothpaste or toothbrushes, plus there was mention of dental supplies. These supplies are the ones that the dentist and hygienists use to perform their job.
	Are used by many Employees	Employees use supplies for various reasons.
Insurance	Idea	Used by patients to pay for the dental services rendered.
Payment	Idea	Money taken from insurance or patients (or both) to pay for services.
Invoice	Document	A document sent to the patient or insurance company explaining how much money is required to pay for services.
	Has Payments	Payments are usually made to cover costs of the invoice (some payments are for other reasons).
Alert	Document	E-mail or phone call made to tell patient of an impending appointment.
Appointment	Event	The event of a patient coming in and having some dental work done.
PatientRecord	Record	All the pertinent information about a patient, much like a patient's chart in any doctor's office.

Figure 4-16 shows how the model has progressed.

You can see, at this point, that the conceptual model has really gelled, and you can get a feel for what the final model might look like. In the next section, we will start adding attributes to the tables, and the model will truly start to take form. It is not 100 percent complete, and you could probably find a few things that you really want to add or change (for example, the fact that Insurance pays Invoice stands out is a definite possibility). However, note that we are trying our best in this phase of the design (certainly in this exercise) to avoid adding value/information to the model. That is part of the process that comes later as you fill in the holes in the documentation that you are given from the client.

www.it-ebooks.info

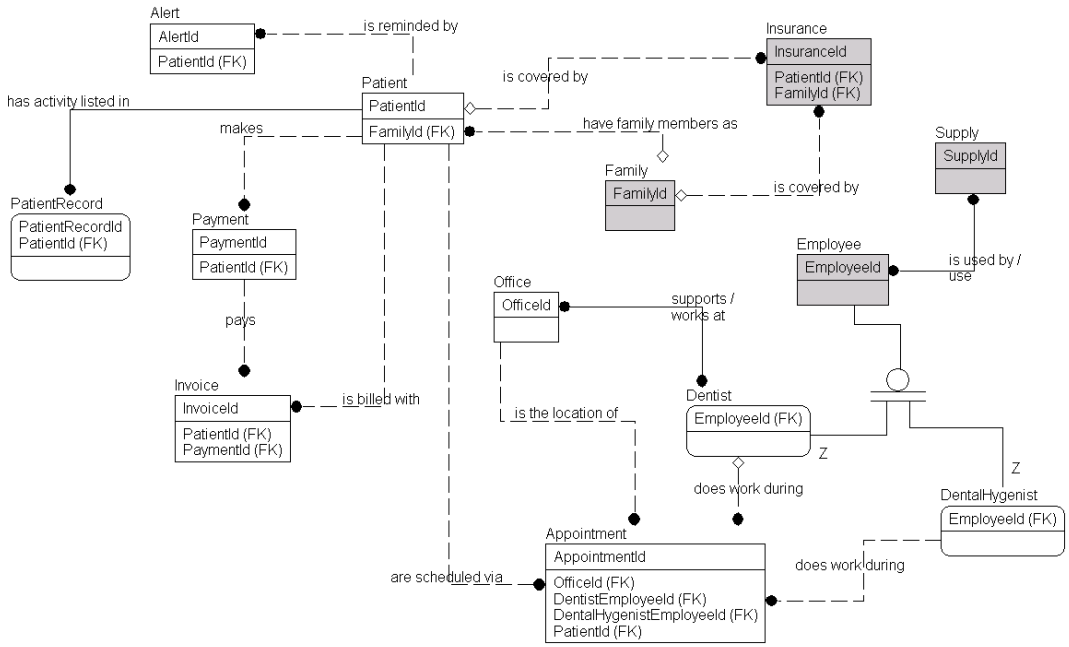


Figure 4-16. *The final conceptual model*

Bear in mind that the key attributes that I have included on this model were used as a method to show lineage. The only time I used any role names for attributes was in the final model, when I related the two subtypes of employee to the appointment entity, as shown in Figure 4-17.

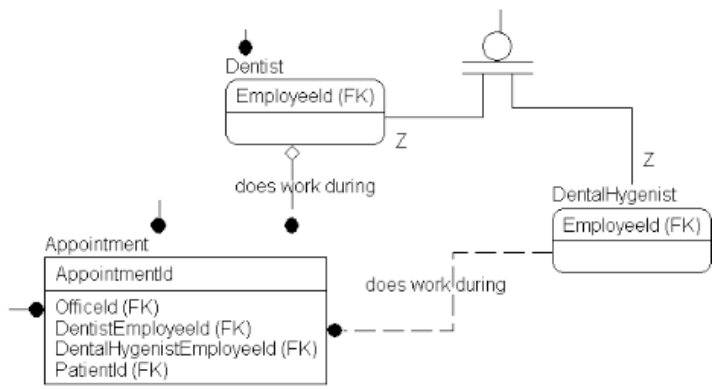


Figure 4-17. *Appointment entity with a role named EmployeeRelationship*

I related the two subtypes to the appointment entity to make it clear what the role of each relationship was for, rather than having the generic `EmployeeId` in the table for both relationships. Again, even the use of any sort of key is not a standard conceptual model construct, but without relationship attributes, the model seems sterile and also tends to hide lineage from entity to entity.

www.it-ebooks.info

Identifying Attributes and Domains

As we start the initial phase of creating a logical model, the goal is to look for items that identify and describe the entity you're trying to represent, or—to put this into more computing-like terms—the properties of your entities. For example, if the entity is a person, attributes might include a driver's license number, Social Security number, hair color, eye color, weight, spouse, children, mailing address, and e-mail address. Each of these things serves to represent the entity in part.

Identifying which attributes to associate with an entity requires a similar approach to identifying the entities themselves. You can frequently find attributes by noting adjectives that are used to describe an entity you have previously found. Some attributes will simply be discovered because of the type of entity they are (person, place, and so on).

Domain information for an attribute is generally discovered at the same time as the attributes, so at this point, you should identify domains whenever you can conveniently locate them. The following is a list of some of the common types of attributes to look for during the process of identifying attributes and their domains:

- *Identifiers*: Any information used to identify a single instance of an entity. This will be loosely analogous to a key, though identifiers won't always make proper keys.
- *Descriptive information*: Information used to describe something about the entity, such as color, amounts, and so on.
- *Locators*: Identify how to locate what the entity is modeling, such as a mailing address, or on a smaller scale, a position on a computer screen.
- *Values*: Things that quantify something about the entity, such as monetary amounts, counts, dates, and so on.

As was true during our entity search, these aren't the only places to look for attributes, but they're a good place to start. The most important thing for now is that you'll look for values that make it clearer what the entity is modeling. Also, it should be noted that all of these have equal merit and value, and groupings may overlap. Lots of attributes will not fit into these groupings (even if all of my example attributes all too conveniently will). These are just a set of ideas to get you help you when looking for attributes.

Identifiers

In this section, we will consider elements used to identify one instance from another. Every entity needs to have at least one identifying attribute or set of attributes. Without attributes, there's no way that different objects can be identified later in the process. These identifiers are likely to end up being used as candidate keys of the entity. For example, here are some common examples of good identifiers:

- *For people*: Social Security numbers (in the United States), full names (not always a perfect identifier), or other IDs (such as customer numbers, employee numbers, and so on).
- *For transactional documents (invoices, bills, computer-generated notices)*: These usually have some sort of externally created number assigned for tracking purposes.
- *For books*: The ISBN numbers (titles definitely aren't unique, not even always by author).
- *For products*: Product numbers for a particular manufacturer (product names aren't unique).
- *For companies that clients deal with*: These are commonly assigned a customer/client number for tracking.
- *For buildings*: Often, a building will be given a name to be referred to.
- *For mail*: The addressee's name and address and the date it was sent.

www.it-ebooks.info

This is not by any means an exhaustive list, but this representative list will help you understand what identifiers mean. Think back to the relational model overview in Chapter 1—each instance of an entity must be unique. Identifying unique natural keys in the data is the first step in implementing a design.

Take care to really discern whether what you think of as a unique item is actually unique. Look at people's names. At first glance, they almost seem unique, and in real life you will use them as keys, but in a database, doing so becomes problematic. For example, there are hundreds of Louis Davidsons in the United States. And thousands, if not millions, of John Smiths are out there! For these cases, you may want to identify in the documentation what I call "likely uniqueness." Is it possible to have two John Smiths as customers? Definitely, but is it extremely likely? That depends on the size of your clientele. In your model and eventually in your applications, you will most likely want to identify data that is not actually a good key (like first and last name) but that is very likely unique. Using this information, the UI might identify likely matching people when you put in first and last name and then ask for a known piece of information rather than expecting that it is a new customer. (In Chapter 8, we will discuss the different ways we can implement uniqueness criteria; for now, it is important to document the cases.)

In our example, the first such example of an identifier is found in this phrase:

> *The client manages a couple of dental offices. One is called the* `Chelsea Office`*, the other the* `Downtown Office`*.*

Almost every case where something is given a name, it's a good attribute to identify the entity, in our case `Name` for `Office`. This makes it a likely candidate for a key because it's unlikely that the client has two offices that it refers to as "Downtown Office," because that would be confusing. So, I add the name attribute to the `Office` entity in the model (shown in Figure 4-18). I'll create a generic domain for these types of generic names, for which I generally choose 60 characters as a reasonable length. This isn't a replacement for validation, because

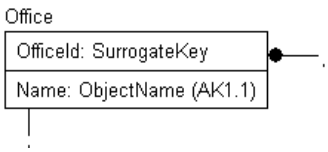


Figure 4-18. *Added the Name attribute to the Office entity*

the client might have specific size requirements for attributes, though most of the time, the client will not really give a thought to lengths, nor care initially until reports are created and the values have to be displayed. I use 60 because that is well over half of the number of characters that can be displayed on a normal document or form:

```
123456789012345678901234567890123456789012345678901234567890
```

The actual default length can easily be changed. That is the point of using domains.

■ **Tip** Report formatting can often vary what your model can handle, but be careful about letting it be the complete guide. If 200 characters are needed to form a good name, use 200, and then create attributes that shorten the name for reports. When you get to testing, if 200 is the maximum length, then all forms, reports, queries, and so on should be tested for the full-sized attribute's size, hence the desire to keep things to a reasonable length.

---

www.it-ebooks.info

When I added the `Name` attribute to the `Office` entity in Figure 4-18, I also set it to require `unique values`, because it would be really awkward to have two offices named the same time, unless you are modeling the dentist/carpentry office for Moe, Larry, and Shemp...

Another identifier is found in this text:

> *Currently the client uses a patient number in its computer system that corresponds to a particular folder that has the patient's records.*

Hence, the system needs a patient number attribute for the `Patient` entity. Again, this is one of those places where querying the client for the specifications of the patient number is a good idea. For this reason, I'll create a specific domain for the patient number that can be tweaked if needed. After further discussion, we learn that the client is using eight-character patient numbers from the existing system (see Figure 4-19).

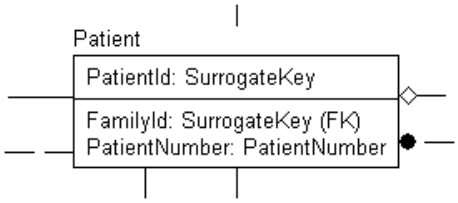


Figure 4-19. *Added the PatientNumber attribute to the Patient entity*

■ **Note** I used the name `PatientNumber` in this entity, even though it included the name of the table as a suffix (something I suggested that should be done sparingly). I did this because it's a common term to the client. It also gives clarity to the name that Number would not have. Other examples might be terms like `PurchaseOrderNumber` or `DriversLicenseNumber`, where the meaning sticks out to the client. No matter what your naming standards, it's generally best to make sure that terms that are common to the client appear as the client normally uses them.

For the most part, it's usually easy to discover an entity's identifier, and this is especially true for the kinds of naturally occurring entities that you find in user-based specifications. Most everything that exists naturally has some sort of way to differentiate itself, although differentiation can become harder when you start to dig deeper.

Of course, in the previous paragraph I said "usually," and I meant it. A common contra-positive to the prior statement about everything being identifiable is things that are managed in bulk. Take our dentist office—although it's easy to differentiate between toothpaste and floss, how would you differentiate between two different tubes of toothpaste? And do you really care? It's probably a safe enough bet that no one cares which tube of toothpaste is given to little Johnny, but this knowledge might be important when it comes to the narcotics that might be distributed. More discussion with the client would be necessary, but my point is that differentiation isn't always simple. During the early phase of logical design, the goal is to do the best you can. Some details like this can become implementation details. For narcotics, we might require a label be printed with a code and maintained for every bottle. For toothpaste, you may have one row and an estimated inventory amount. In the former, the key might be the code you generate and print, and in the latter, the name "toothpaste" might be the key, regardless of the actual brand of toothpaste sample.

www.it-ebooks.info

Descriptive Information

Descriptive information refers to the common types of adjectives used to describe things that have been previously identified as entities and will usually point directly to an attribute. In our example, different types of supplies are identified, namely, sample and dental:

> *. . . their supplies, such as* `sample toothpastes, toothbrushes, and floss, as well as` ***dental supplies****.*

Another thing you can identify is the possible domain of an attribute. In this case, the attribute is "Type of Supply," and the domain seems to be "Sample" and "Dental." Hence, I create a specific special domain: `SupplyType` (see Figure 4-20).

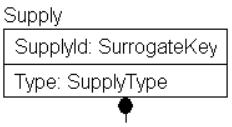


Figure 4-20. *Added the Type attribute to the Supply entity*

Locators

The concept of a locator is not unlike the concept of a key, except that instead of talking about locating something within the electronic boundaries of our database, the locator finds the geographic location, physical position, or even electronic location of something.

For example, the following are examples of locators:

- *Mailing address*: Every address leads us to some physical location on Earth, such as a mailbox at a house or even a post office box in a building.
- *Geographical references*: These are things such as longitude and latitude or even textual directions on how to get to some place.
- *Phone numbers*: Although you can't always pinpoint a physical location using the phone number, you can use it to locate a person.
- *E-mail addresses*: As with phone numbers, you can use these to locate and contact a person.
- *Web sites, FTP sites, or other assorted web resources*: You'll often need to identify the web site of an entity or the URL of a resource that's identified by the entity; such information would be defined as attributes.
- *Coordinates of any type*: These might be a location on a shelf, pixels on a computer screen, an office number, and so on.

The most obvious location we have in our example is an office, going back to the text we used in the previous section:

> *The client manages a couple of dental offices. One is called the* `Chelsea Office`*, the other the* `Downtown Office`*.*

www.it-ebooks.info

It is reasonably clear from the names that the offices are not located together (like in the same building that has 100 floors, where one office is a more posh environment or something), so another identifier we should add is the building address. Buildings will be identified by its geographic location because a nonmoving target can always be physically located with an address or geographic coordinates. Figure 4-21 shows office entity after adding the address:

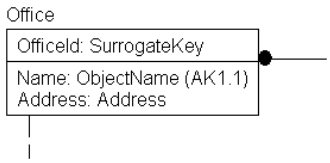


Figure 4-21. *Added an Address attribute to the Office entity*

Each office can have only one address that identifies its location, so the address initially can go directly in the office entity. Also important is that the domain for this address be a physical address, not a post office box.

Places aren't the only things you can locate. People are locatable as well. In this loose definition, a person's location can be a temporary location or a contact that can be made with the locator, such as addresses, phone numbers, or even something like GPS coordinates, which might change quite rapidly. In this next example, there are three typical locators:

> *. . . have an address, a phone number (home, mobile, and/or office), and optionally an e-mail address associated with each family, and possibly patient if the client desires...*

Most customers, in this case the dental victims—er, patients—have phone numbers, addresses, and/or e-mail address attributes. The dental office uses these to locate and communicate with the patient for many different reasons, such as billing, making and canceling appointments, and so on. Note also that often families don't live together, because of college, divorce, and so on, but you might still have to associate them for insurance and billing purposes. From these factors you get these sets of attributes on families and patients; see Figure 4-22.

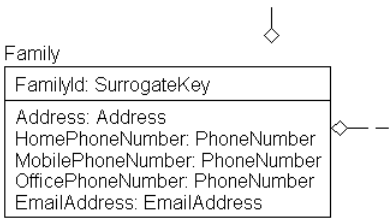


Figure 4-22. *Added location-specific attributes to the Family entity*

The same is found for the patients, as shown in Figure 4-23.

This is a good place to reiterate one of the major differences between a column that you are intending to implement and an attribute in your early modeling process. An attribute needn't follow any specific requirement for its shape. It might be a scalar value; it might be a vector, and it might be a table in and of itself. A column in your the physical database you implement needs to fit a certain mold of being a scalar or fixed vector and nothing else. In logical modeling, the goal is documentation of structural needs and moving closer to what you will implement. The normalization process completes the process of shaping all of the attributes into the proper shape for implementation in our relational database.

www.it-ebooks.info

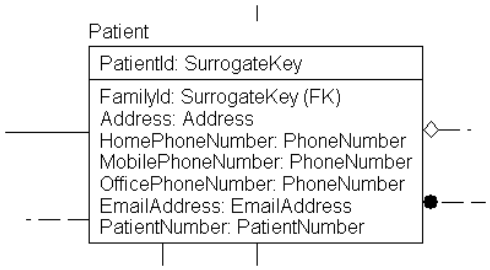


Figure 4-23. *Added location-specific attributes to the Patient entity*

It's enough at this early phase of modeling to realize that when users see the word "address" in the context of this example, they think of a generic address used to locate a physical location. In this manner, you can avoid any discussion of how the address is implemented, not to mention all the different address formats that might need to be dealt with when the address attribute is implemented later in the book.

Values

Numbers are some of the most powerful attributes, because often, math is performed with them to get your client paid, or to calculate or forecast revenue. Get the number of dependents wrong for a person, and his or her taxes will be messed up. Or get your wife's weight wrong in the decidedly wrong direction on a form, and she might just beat you with some sort of cooking device (sad indeed).

Values are generally numeric, such as the following examples:

- *Monetary amounts*: Financial transactions, invoice line items, and so on
- *Quantities*: Weights, number of products sold, counts of items (number of pills in a prescription bottle), number of items on an invoice line item, number of calls made on a phone, and so on
- *Other*: Wattage for light bulbs, size of a TV screen, RPM rating of a hard disk, maximum speed on tires, and so on

Numbers are used all around as attributes and are generally going to be rather important (not, of course, to minimize the value of other attributes!). They're also likely candidates to have domains chosen for them to make sure their values are reasonable. If you were writing a package to capture tax information about a person, you would almost certainly want a domain to state that the count of dependents must be greater than or equal to zero. You might also want to set a likely maximum value, such as 30. It might not be a hard and fast rule, but it would be a sanity check, because most people don't have 30 dependents (well, most sane people, before, or certainly not after!). Domains don't have to be hard and fast rules at this point (only the hard and fast rules will likely end up as database constraints, but they have to be implemented somewhere, or users can and will put in whatever they feel like at the time).

In our example paragraphs, there's one such attribute:

The client manages a `couple of dental offices`.

The question here is what attribute this would be. In this case, it turns out it won't be a numeric value, but instead some information about the cardinality of the dental `Office` entity. There would be others in the model once we dug deeper into invoicing and payments, but I specifically avoided having monetary values to keep things simple in the model.

www.it-ebooks.info

Relationship Attributes

Every relationship that's identified might imply bits of data to support it. For example, consider a common relationship such as `Customer pays Invoice`. That's simple enough; this implies a relationship between the `Customer` entity and the `Invoice` entity. But the relationship implies that an invoice needs to be paid; hence (if you didn't know what an invoice was already), it's now known that an invoice has some form of amount attribute.

As an example in our database, in the relationship `Employees use Supplies for various reasons`, the "for various reasons" part may lead us to the related-information type of attribute. What this tells us is that the relationship isn't a one-to-many relationship between `Person` and `Supplies`, but it is a many-to-many relationship between them. However, it does imply that an additional entity may later be needed to document this fact, since it's desirable to identify more information about the relationship.

■ **Tip** Don't fret too hard that you might miss something essential early in the design process. Often, the same entity, attribute, or relationship will crop up in multiple places in the documentation, and your clients will also recognize many bits of information that you miss as you review things with the over and over until you are happy with your design and start to implement.

A List of Entities, Attributes, and Domains

Figure 4-24 shows the logical graphical model as it stands now and Table 4-3 shows the entities, along with descriptions and column domains. The attributes of an entity are indented within the Entity/Attribute column

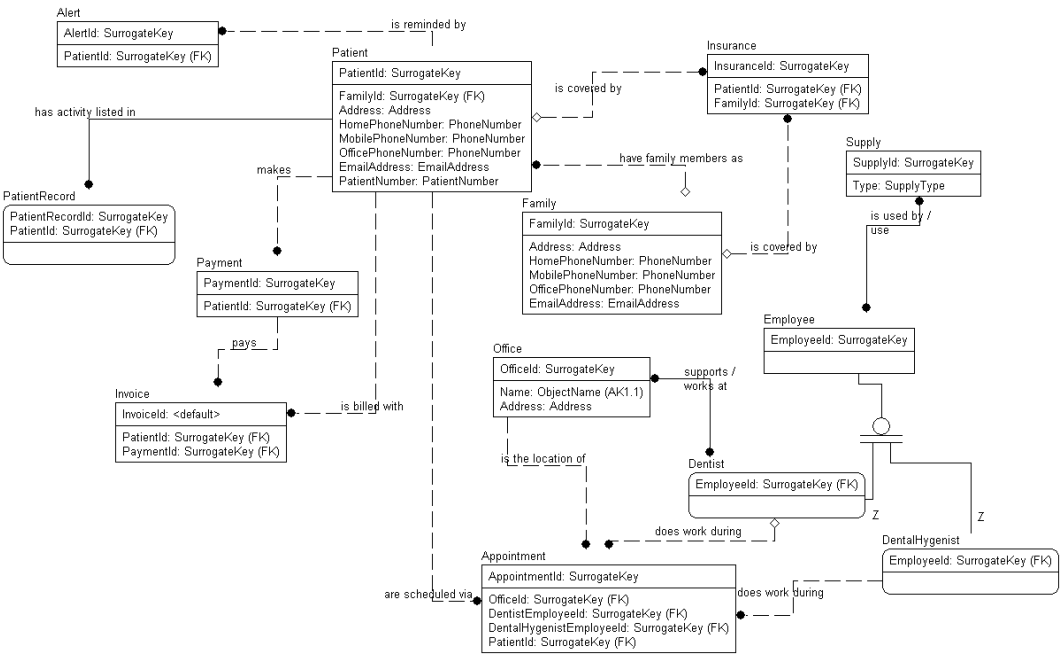


Figure 4-24. *Graphical model of the patient system so far*

www.it-ebooks.info

Table 4.3. *Final Model for the Dental Office Example*

Entity/Attribute	Description	Column Description	Column Domain
`Patient`	The people who are the customers of the dental office. Services are performed, supplies are used, and patients are billed for them.		
	`PatientNumber`	Used to identify a patient's records in the computer	Unknown, generated by the current computer system
	`HomePhoneNumber`	Phone number to call patient at home	Any valid phone number
	`MobilePhoneNumber`	Phone number to call patient away from home	Any valid phone number
	`OfficePhoneNumber`	Phone number to call patient during work hours *(Note: Do we need to know work hours for the patient?)*	Any valid phone number
	`Address`	Postal address of the family	Any valid address
	`EmailAddress`	Electronic mail address of the family	Any valid e-mail address
`Family`	Groups of persons who are associated, likely for insurance purposes.		
	`HomePhoneNumber`	Phone number to call patient at home	Any valid phone number
	`MobilePhoneNumber`	Phone number to call patient away from home	Any valid phone number
	`OfficePhoneNumber`	Phone number to call patient during work hours (*Note: Do we need to know work hours for the patient?*)	Any valid phone number
	`Address`	Postal address of the family	Any valid address
	`EmailAddress`	Electronic mail address of the family	Any valid e-mail address
	`FamilyMembers`	Patients that make up a family unit	Any patients (*Note: Can a patient be a member of only one family?*)
`Dentist`	Persons who do the most important work at the dental office. Several dentists work for the client's practice.		

(continued)

www.it-ebooks.info

Table 4.3. *(continued)*

Entity/Attribute	Description	Column Description	Column Domain
DentalHygienist	People who do the basic work for the dentist. There are quite a few more hygienists than dentists. (*Note: Check with client to see if there are guidelines for the number of hygienists per dentist. Might be needed for setting appointments.*)		
Employee	Any person who works at the dental office. Dentists and hygienists are clearly types of employees.		
Office	Locations where the dentists do their business. They have multiple offices to deal with and schedule patients for.		
	Address	Physical address where the building is located	Address that is not a PO box
	Name	The name used to refer to a given office	Unique
Supply	Examples given were sample items, such as toothpaste or toothbrushes; plus, there was mention of dental supplies. These supplies are the ones that the dentist and hygienists use to perform their jobs.		
	Type	Classifies supplies into different types	"Sample" or "Dental" identified

Implementation modeling note: Log any changes to sensitive or important data. The relationship between employees and supplies will likely need additional information to document the purpose for the usage.

www.it-ebooks.info

(I've removed the relationships found in the previous document for clarity). Note I've taken the list a bit further to include all the entities I've found in the paragraphs and will add the attributes to the model after the list is complete.

Table 4-3 lists a subset of the descriptive metadata.

■ **Tip** Consider carefully the use of the phrase "any valid" or any of its derivatives. The scope of these statements needs to be reduced to a reasonable form. In other words, what does "valid" mean? The phrases "valid dates" indicates that there must be something that could be considered invalid. This, in turn, could mean the "November 31st" kind of invalid or that it isn't valid to schedule an appointment during the year 1000 BC. Common sense can take us a long way, but computers seriously lack common sense without human intervention.

Note that I added another many-to-many relationship between `Appointment` and `Supply` to document that supplies are used during appointments. Figure 4-25 shows the final graphical model that we can directly discern from the slight description we were provided:

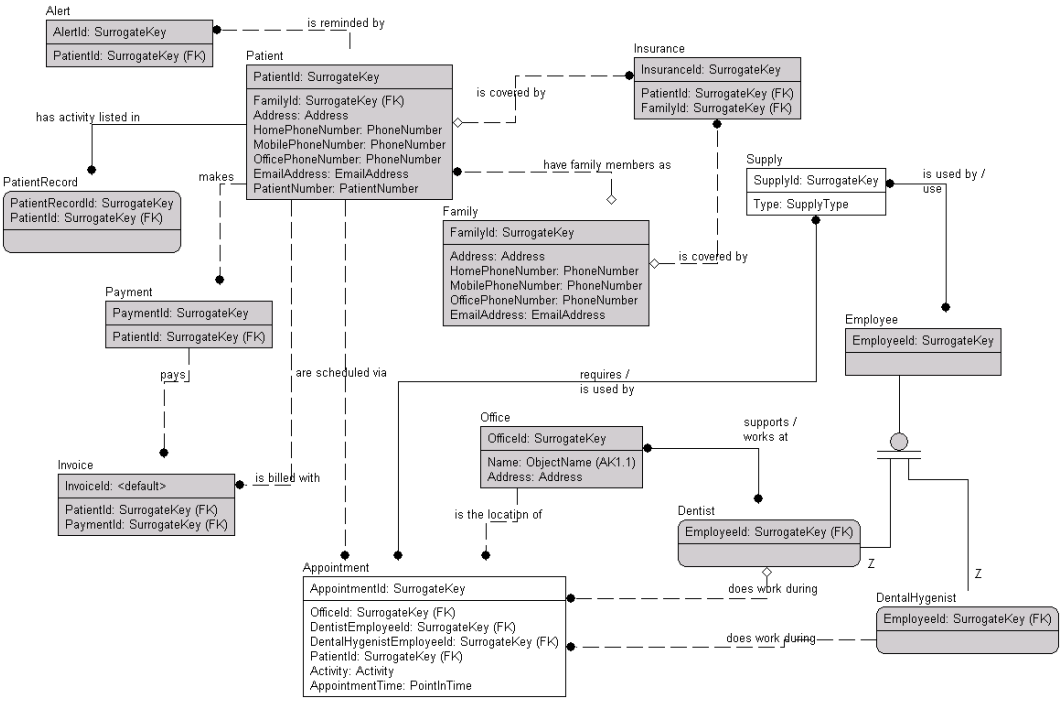


Figure 4-25. *Model with all entities, attributes, and relationships that were found directly in the model*

At this point, the entities and attributes have been defined. Note that nothing has been added to the design that wasn't directly implied by the single requirement artifact we started with. When doing this kind of activity in a real setting, all the steps of finding entities, relationships, and attributes would likely be handled at one time. In this chapter, I've performed the steps in a deliberate, step-by-step process only to focus on one at a time to make

www.it-ebooks.info

the parts of the process clearer. If this had been a real design session, whenever I found something to add to the model, I would have added it immediately.

It might also be interesting to note that the document is now several pages long—all from analyzing three small paragraphs of text. When you do this in a real project, the resulting document will be much larger, and there will likely be quite a bit of redundancy in much of the documentation.

Identifying Business Rules

Business rules can be defined as statements that govern and shape business behavior. Depending on an organization's methodology, these rules can be in the form of bulleted lists, simple text diagrams, or other formats (too often they are stored in a key employee's head). A business rule's existence doesn't imply the ability to implement it in the database at this point in the process. The goal is to get down all data-oriented rules for use later in the process.

When defining business rules, there might be some duplication of rules and attribute domains, but this isn't a real problem at this point. Get as many rules as possible documented, because missing business rules will hurt you more than missing attributes, relationships, or even tables. You'll frequently find new tables and attributes when you're implementing the system, usually out of necessity, but finding new business rules at a late stage can wreck the entire design, forcing an expensive rethink or an ill-advised "kludge" to shoehorn them in.

Recognizing business rules isn't generally a difficult process, but it is time-consuming and fairly tedious. Unlike entities, attributes, and relationships, there's no straightforward, specific grammar-oriented clue for identifying all the business rules. In essence, everything we have done so far in this chapter is, in fact, just specialized versions of business rules.

However, my general practice when I have to look for business rules is to read documents line by line, looking for sentences including language such as "once . . . occurs," ". . . have to . . . ," ". . . must . . . ," ". . . will . . . ," and so on. Unfortunately for you, documents don't usually include every business rule, and it is just as great a folly to expect that your clients will remember all of them right off the top of their heads. You might look through a hundred or a thousand invoices and not see a single instance where a client is credited money, but this doesn't mean it never happens. In many cases, you have to mine business rules from three places:

- *Old code*: It's the exception, not the rule, that an existing system will have great documentation. Even the ones that start out with wonderful system documentation tend to have their documentation grow worse and worse as time grows shorter and client desires grow. It isn't uncommon to run into poorly written spaghetti code that needs to be analyzed.
- *Client experience*: Using human memory for documentation can be as hard as asking teenagers what they did the night before. Forgetting important points of the store, or simply making up stuff that they think you want to hear is just part of human nature. I've already touched on how difficult it is to get requirements from users, but when you get into rules, this difficulty grows by at least an order of magnitude because most humans don't think in details, and a good portion of the business-rules hunt is about minute details.
- *Your experience*: Or at least the experience of one member of your team. Like the invoice example, you might ask questions like "do you ever…" to jog the customer's memory. If you smell rotten cheese, it is usually not because it is supposed to smell that way.

If you're lucky, you'll be blessed by a business analyst who will take care of this process, but in a lot of cases the business analyst won't have the programming experience to think in code-level details, and to ferret out subtle business rules from code, so a programmer may have to handle this task. That's not to mention that it's hard to get to the minute details until you understand the system, something you can do only by spending lots of

www.it-ebooks.info

time thinking, considering, and digesting what you are reading. Rare is the occasion going to be afforded you to spend enough time to do a good job.

In our "snippet of notes from the meeting" example, a few business rules need to be defined. For example, I've already discussed the need for a customer number attribute but was unable to specify a domain for the customer number. Take the following sentence:

For each appointment, the client needs to have everything documented that went on . . .

From it, you can derive a business rule such as this:

For every appointment, it is required to document every action on the patient's chart so it can be charged.

Note that this rule brings up the likelihood that there exists yet another attribute of a patient's chart—`Activity`—and another attribute of the activity—`ActivityPrices`. This relationship between `Patient`, `PatientRecord`, `Activity`, and `ActivityPrices` gives you a feeling that it might be wrong. It would be wrong to implement it in code this way, very wrong. Normalization corrects this sort of dependency issue, and it's logical that there exists an entity for activities with attributes of `name` and `price` that relate back to the `PatientRecord` entity that has already been created. Either way is acceptable before calling an end to the modeling process, as long as it makes sense to the readers of the documents. I'll go ahead and add an `Activity` entity with a name and a price for this requirement.

Another sentence in our example suggests a further possible business rule:

The dentists might spend time at each of the offices throughout the week.

Obviously, a doctor cannot be in two different locations at one time. Hence, we have the following rule:

Doctors must not be scheduled for appointments at two locations at one time.

Another rule that's probably needed is one that pertains to the length of time between appointments for doctors:

The length of time between appointments for dentists at different offices can be no shorter than X.

Not every business rule will manifest itself within the database, even some that specifically deal with a process that manages data. For example, consider this rule:

Invoices should be sent within one week after the appointment.

This is great and everything, but what if it takes a week and a day, or even two weeks? Can the invoice no longer be sent to the patient? Should there be database code to chastise the person if someone was sick, and it took a few hours longer than a week? No; although this seems much like a rule that could be implemented in the database, it isn't. This rule will be given to the people doing system documentation and UI design for use when designing the rest of the system. The other people working on the design of the overall system will often provide us with additional entities and attributes.

The specifics of some types of rules will be dealt with later in Chapters 6 and 7, as we implement various types of tables and integrity constraints.

www.it-ebooks.info

Identifying Fundamental Processes

A process is a sequence of steps undertaken by a program that uses the data that has been identified to do something. It might be a computer-based process, such as "process daily receipts," where some form of report is created, or possibly a deposit is created to send to the bank. It could be something manual, such as "creating new patient," which details that first the patient fills out a set of forms, then the receptionist asks many of the same questions, and finally, the nurse and doctor ask the same questions again once arriving in the room. Then, some of this information is keyed into the computer after the patient leaves so the dental office can send a bill.

You can figure out a lot about your client by studying their processes. Often, a process that you guess should take two steps and ten minutes can drag on for months and months. The hard part will be determining why. Is it for good, often security oriented reasons? Or is the long process the result of historical inertia? There are reasons for every bizarre behavior out there, and you may or may not be able to figure out why it is as it is and possibly make changes. At a minimum, the processes will be a guide to some of the data you need, when it is required, and who uses the data in the organization operationally.

As a reasonable manual-process example, consider the process of getting a driver's license (at least in Tennessee for a new driver. There are other processes that are followed if you come from another state, are a certain age, are not a citizen, etc.):

1. Fill in learner's permit forms.
2. Obtain learner's permit.
3. Practice.
4. Fill in license forms.
5. Pass eye exam.
6. Pass written exam.
7. Pass driving exam.
8. Have picture taken.
9. Receive license.

Processes might or might not have each step well enumerated during the logical design phase, and many times, a lot of processes are fleshed out during the physical database implementation phase in order to accommodate the tools that are available at the time of implementation. I should mention that most processes have some number of *process rules* associated with them (which are business rules that govern the process, much like those that govern data values). For example, you must complete each of those steps (taking tests, practicing driving, and so on) before you get your license. Note that some business rules are also lurking around in here, because some steps in a process might be done in any order. For example you could have the written exam before the eye exam and the process would remain acceptable, while others must be done in order. Like if you received the license without passing the exams, which would be kind of stupid, even for a process created by a bureaucracy.

In the license process, you have not only an explicit order that some tasks must be performed but other rules too, such as that you must be 15 to get a learner's permit, you must be 16 to get the license, you must pass the exam, practice must be with a licensed driver, and so on (and there are even exceptions to some of these rules, like getting a license earlier if you are a hardship case!) If you were the business analyst helping to design a driver's license project, you would have to document this process at some point.

Identifying processes (and the rules that govern them) is relevant to the task of data modeling because many of these processes will require manipulation of data. Each process usually translates into one or more queries or stored procedures, which might require more data than has been specified, particularly to store state information throughout the process.

In our example, there are a few examples of such processes:

www.it-ebooks.info

The client needs the system to manage its patients and appointments . . .

This implies that the client needs to be able to make appointments, as well as manage the patients—presumably the information about them. Making appointments is one of the most central things our system will do, and you will need to answer questions like these: What appointments are available during scheduling? When can appointments be made?

This is certainly a process that you would want to go back to the client and understand:

. . . and then invoice the patient's insurance, if he or she has insurance (otherwise the patient pays).

I've discussed invoices already, but the process of creating an invoice might require additional attributes to identify that an invoice has been sent electronically or printed (possibly reprinted). Document control is an important part of many processes when helping an organization that's trying to modernize a paper system. Note that sending an invoice might seem like a pretty inane event—press a button on a screen, and paper pops out of the printer. All this requires is selecting some data from a table, so what's the big deal? However, when a document is printed, we might have to record the fact that the document was printed, who printed it, and what the use of the document is. We might also need to indicate that the documents are printed during a process that includes closing out and totaling the items on an invoice. The most important point here is that you shouldn't make any major assumptions.

Here are other processes that have been listed:

- *Track and manage dentists and hygienists*: From the sentence, "The system needs to track and manage several dentists, and quite a few dental hygienists who the client needs to allocate to each appointment as well."
- *Track supplies*: From "The client has had problems in the past keeping up with when it's about to run out of supplies, and wants this system to take care of this for both locations. For the dental supplies, we need to track usage by employee, and especially any changes made in the database to the patient records."
- *Alert patient*: From "alerting the patients when their appointments occur, either by e-mail or by phone . . ."

Each of these processes identifies a unit of work that you must deal with during the implementation phase of the database design procedure.

The Intermediate Version of the Logical Model

In this section, I'll briefly cover the steps involved in completing the task of establishing a working set of documentation. There's no way that we have a complete understanding of the documentation needs now, nor have we yet discovered all the entities, attributes, relationships, business rules, and processes that the final system will require. However, the better the job you do, the easier the rest of the process of designing and implementing the final system will be.

On the other hand, be careful, because there's a sweet spot when it comes to the amount of design needed. After a certain point, you could keep designing and make little—if any—progress. This is commonly known as *analysis paralysis*. Finding this sweet spot requires experience. Most of the time, too little design occurs, usually because of a deadline that was set without any understanding of the realities of building a system. On the other

www.it-ebooks.info

hand, without strong management, I've found that I easily get myself into analysis paralysis (hey, this book focuses on design for a reason; to me it's the most fun part of the project).

The final steps of this discovery phase remain (the initial discovery phase anyhow, because you'll have to go back occasionally to this process to fill in gaps that were missed the first time). There are a few more things to do, if possible, before starting to write code:

1. Identify obvious additional data needs.
2. Review the progress of the project with the client.
3. Repeat the process until you're satisfied and the client is happy and signs off on what has been designed.

These steps are part of any system design, not just the data-driven parts.

Identifying Obvious Additional Data Needs

Up until this point, I've been reasonably careful not to broaden the information that was included from the discovery phase. The purpose has been to achieve a baseline to our documentation, staying faithful to the piles of documentation that were originally gathered. Mixing in our new thoughts prior to agreeing on what was in the previous documentation can be confusing to the client, as well as to us. However, at this point in the design, you need to change direction and begin to add the attributes that come naturally. Usually, there's a fairly large set of obvious attributes and, to a lesser extent, business rules that haven't been specified by any of the users or initial analysis. Make sure any assumed entities, attributes, relationships, and so on stand out from what you have gotten from the documentation.

For the things that have been identified so far, go through and specify additional attributes that will likely be needed. For example, take the `Patient` entity, as shown in Table 4-4.

Table 4.4. *Completed Patient Entity*

Entity	Description	Domain
`Patient`	The people who are the customers of the dentist office. Services are performed, supplies are used, and they are billed for them.	
Attributes		
`PatientNumber`	Used to identify a patient's records, in the current computer system	Unknown; generated by computer and on the chart?
`Insurance`	Identifies the patient's insurance carrier.	Unknown *(Note: Check for common formats used by insurance carriers, perhaps?)*
Relationships		
	Has `Alerts`	Alerts are sent to patients to remind them of their appointments.
	Has `Appointments`	Appointments need to have one patient.
	Has `Invoices`	Patients are charged for appointments via an invoice.
	Makes `Payment`	Patients make payments for invoices they receive.

www.it-ebooks.info

The following additional attributes are almost certainly desirable:

- *Name*: The patient's full name is probably the most important attribute of all.
- *Birth date*: If the person's birthday is known, a card might be sent on that date. This is probably also a necessity for insurance purposes.

You could certainly add more attributes for the `Patient` entity, but this set should make the point clearly enough. There might also be additional tables, business rules, and so on, to recommend to the client. In this phase of the design, document them, and add them to your lists.

One of the main things to do is to identify when you make any large changes to the customer's model. In this example, the client might not want to keep up with the birth dates of its patients (though as noted, it's probably an insurance requirement that wasn't initially thought of).

The process of adding new stuff to the client's model based on common knowledge is essential to the process and will turn out to be a large part of the process. Rarely will the analyst think of everything.

Review with the Client

Once you've finished putting together this first-draft document, it's time to meet with the client to explain where you've gotten to in your design and have the client review every bit of this document. Make sure the client understands the solution that you're beginning to devise.

It's also worthwhile to follow or devise some form of sign-off process or document, which the client signs before you move forward in the process. In some cases, your sign-off documents could well be legally binding documents and will certainly be important should the project go south later for one reason or another. Obviously, the hope is that this doesn't happen, but projects fail for many reasons, and a good number of them are not related to the project itself. It's always best if everyone is on the same page, and this is the place to do that.

Repeat Until the Customer Agrees with Your Model

It isn't likely you'll get everything right in this phase of the project. The most important thing is to get as much correct as you can and get the customer to agree with this. Of course, it's unlikely that the client will immediately agree with everything you say, even if you're the greatest data architect in the world. It is also true that often the client will know what they want just fine but cannot express it in a way that gets through your thick skull. In either event, it usually takes several attempts to get the model to a place where everyone is in agreement. Each iteration should move you and the client closer to your goal.

There will be many times later in the project that you might have to revisit this part of the design and find something you missed or something the client forgot to share with you. As you get through more and more iterations of the design, it becomes increasingly important to make sure you have your client sign off at regular times; you can point to these documents when the client changes his or her mind later.

If you don't get agreement, often in writing or in a public forum, such as a meeting with enough witnesses, you can get hurt. This is especially true when you don't do an adequate job of handling the review and documentation process and there's no good documentation to back up your claim versus the clients. I've worked on consulting projects where the project was well designed and agreed on but documentation of what was agreed wasn't done too well (a lot of handshaking at a higher level to "save" money). As time went by and many thousands of dollars were spent, the client reviewed the agreement document, and it became obvious that we didn't agree on much at all. Needless to say, that whole project worked out about as well as hydrogen-filled, thermite coated dirigibles.

www.it-ebooks.info

■ **Note** I've been kind of hard on clients in this chapter, making them out to be conniving folks who will cheat you at the drop of a hat. This is seldom the case, but it takes only one. The truth is that almost every client will appreciate you keeping him or her in the loop and getting approval for the design at reasonable intervals, because clients are only as invested in the process as they have to be. You might even be the fifteenth consultant performing these interviews because the previous 14 were tremendous failures.

Best Practices

The following list of some best practices can be useful to follow when doing conceptual and logical modeling:

- *Be **patient***: A great design comes from not jumping the gun and starting to get ahead of the process. I present the process in this book, as I do to try to encourage you to follow a reasonably linear process rather than starting out with a design looking for a problem to solve with it.
- *Be diligent*: Look through everything to make sure that what's being said makes sense. Be certain to understand as many of the business rules that bind the system as possible before moving on to the next step. Mistakes made early in the process can mushroom later.
- *Document*: The point of this chapter has been just that—document every entity, attribute, relationship, business rule, and process identified (and anything else you discover, even if it won't fit neatly into one of these buckets). The format of the documentation isn't really all that important, only that the information is there, that it's understandable by all parties involved, and that it will be useful going forward toward implementation.
- *Communicate*: Constant communication with clients is essential to keep the design on track. The danger is that if you start to get the wrong idea of what the client needs, every decision past that point might be wrong. Get as much face time with the client as possible.

■ **Note** This mantra of "review with client, review with client, review with client" is probably starting to get a bit old at this point. This is one of the last times I'll mention it, but it's so important that I hope it has sunk in.

Summary

In this chapter, I've presented the process of discovering the structures that should eventually make up a dental-office database solution. We've weeded through all the documentation that had been gathered during the information-gathering phase, doing our best not to add our own contributions to the solution until we processed all the initial documentation, so as not to add our personal ideas to the solution. This is no small task; in our initial example, we had only three paragraphs to work with, yet we ended up with quite a few pages of documentation from it.

It is important to be diligent to determine what kind of building you are building so can create the right kind of foundation. Once you have a firm foundation to build from, the likelihood improves that the database you build on it will be solid and the rest of the process has a chance. If the foundation is shoddy, the rest of the system that gets built will likely be the same. The purpose of this process is to distill as much information as

www.it-ebooks.info

possible about what the client wants out of the system and put it into the conceptual and logical model in order to understand the user's needs.

Once you have as much documentation as possible from the users, the real work begins. Through all this documentation, the goal is to discover as many of the following as possible:

- Entities and relationships
- Attributes and domains
- Business rules that can be enforced in the database
- Processes that require the use of the database

From this, a conceptual and logical data model will emerge that has many of the characteristics that will exist in the actual implemented database. Pretty much all that is left after this is to mold the design into a shape that fits the needs of the RDBMS to provide maximum ease of use. In the upcoming chapters, the database design will certainly change from the model we have just produced, but it will share many of the same characteristics and will probably not be so different that even the nontechnical layperson who has to approve your designs will understand.

In Chapters 6 and 7, we will apply the skills covered in this chapter for translating requirements to a data model and those for normalization from the next chapter to produce portions of data models that demonstrate the many ways you can take these very basic skills and create complex, interesting models.

www.it-ebooks.info

CHAPTER 5

Normalization

Nobody realizes that some people expend tremendous energy merely to be normal.

—Albert Camus

By now, you should have the conceptual and logical model created that covers the data requirements for your database system. As we have discussed over the previous four chapters, our design so far needn't follow any strict format or method. It didn't have to be implementable, but it did have to cover the requirements that are data related. In some cases, the initial creator of the logical model may not even be a database professional, but a client who starts the model. Changes to the logical model meant changes to the requirements of the system.

Now, we come to the most hyped topic in all relational databasedom: *normalization*. It is where the theory meets reality, and everything starts to pay off. We have gathered semistructured piles of data into a model in a manner that is natural to people, but we are not quite ready to start implementing. The final preimplementation step is to take the entities and attributes that have been discovered during the early modeling phases and refine them into tables for implementation in a relational database system. The process does this by removing redundancies and shaping the data in the manner that the relational engine desires to work with it. Once you are done with the process, working with the data will be more natural using the set-based language SQL.

SQL is a language designed to work with atomic values. In computer science terms, *atomic* means that a value cannot (or more reasonably should not) be broken down into smaller parts. Our eventual goal will be to break down the piles of data we have identified into values that are atomic, that is, broken down to the lowest form that will need to be accessed in Transact SQL (T-SQL) code.

The phrase "lowest form" can be dangerous for newbies, because one must resist the temptation to go too far. What "too far" actually means should be obvious by the end of this chapter. One analogy is to consider splitting apart hydrogen and oxygen from water. You can do that, but if you split the hydrogen atom in half, problems will occur. The same is true for columns in a database. At the correct atomic level, your T-SQL will work wonderfully with less tuning, and at the wrong level (too much or too little), you will find yourself struggling against the design.

If you are truly new at database design, you may have heard that once you learn normalization you are going to be the king biscuit database designer. Think, instead, of normalization more like the end of boot camp for young soldiers. When they get out of boot camp, they are highly trained but barely experienced. Understand the theory of how defend your nation is one thing; dodging the bullets of attacking barbarian hordes is quite a different thing.

Then too, you may have heard the term "normalization" used like a bad word by a developer and sometimes rightfully so, based on how people actually do it. The problem usually lies in the approach of the person using normalization to organize the databases, as many people think of normalization as a purely academic activity,

www.it-ebooks.info

almost as if it was a form of ritualistic sacrifice to Codd. Normalization's purpose is to shape the data to match the actual, realistic needs of the users with the cold hard facts of how the relational engine works.

Some joke that if most developers had their way, every database would have exactly one table with two columns. The table would be named "object," and the columns would be "pointer" and "blob." But this desire is usurped by a need to satisfy the customer with searched, reports, and consistent results from their data being stored.

The process of normalization is based on a set of levels, each of which achieves a level of correctness or adherence to a particular set of "rules." The rules are formally known as *forms*, as in the *normal forms*. Quite a few normal forms have been theorized and postulated, but I'll focus on the four most important, commonly known, and often applied. I'll start with First Normal Form (1NF), which eliminates data redundancy (such as a name being stored in two separate places), and continue through to Fifth Normal Form (5NF), which deals with the decomposition of ternary relationships. (One of the normal forms I'll present isn't numbered; it's named for the people who devised it.) Each level of normalization indicates an increasing degree of adherence to the recognized standards of database design. As you increase the degree of normalization of your data, you'll naturally tend to create an increasing number of tables of decreasing width (fewer columns).

■ **Note** If your nerd sense is tingling because I said I would cover four normal forms and one of them is the Fifth Normal Form, you aren't wrong. One of the normal forms actually covers two other numbered forms.

In this chapter, I will look at the different normal forms defined not so much by their numbers, but by the problems they were designed to solve. For each, I will include examples, the programming anomalies they help you avoid, and the telltale signs that your relational data is flouting that particular normal form. It might seem out of place to show programming anomalies at this point, since the early chapters of the book are specifically aligned to the preprogramming design, but it can help reconcile to the programming mind what having data in a given normal form can do to make the tables easier to work in SQL. Finally, I'll wrap up with an overview of some normalization best practices.

The Process of Normalization

The process of normalization is really quite straightforward: take entities that are complex and extract simpler entities from them with the goal of ending up with entities that express fewer concepts than before. The process continues until we produce a model such that, in the implementation, every table in the database will represent one thing and every column describes that thing. This will become more apparent throughout the chapter as I work through the different normal forms.

I'll break down normalization into three general categories:

- Table and column shape
- Relationships between columns
- Multivalued and join dependencies in tables

Note that the conditions mentioned for each step should be considered for every entity you design, because each normal form is built on the precept that the lower forms have been complied with. The reality is that most designs, and even fewer implementations, do not meet any of the normal forms perfectly. Just like breaking down and having a donut on a diet doesn't mean you should stop dieting, imperfections are a part of the reality of database design. As an architect, you will strive for perfection, but it is largely impossible to achieve, if for no other reason than the fact that users' needs change frequently (and impatient project managers demand table counts and completion dates far in advance of it being realistic to make such estimates), to say the least.

www.it-ebooks.info

In Agile projects, I simply try to do the best I can to meet the demanding schedule requirements, but minimally, I try to at least document and know what the design should be because the perfect design matches the real world in a way that makes it natural to work with (if sometimes a bit tedious). Design and implementations will always be a trade off with your schedule, but the more you know about what correct is, the more likely you will be able to eventually achieve it, regardless of artificial schedule milestones.

Table and Column Shape

The first step in the process of producing a normalized database is to deal with the "shape" of the data. If you ignored the rest of this book, it is minimally important to understand how the relational engine wants the data shaped. If you recall back in Chapter 1, when we covered Codd's 12 Rules, the first two are the basis for the definition of a table. The first stated that data was to be represented by values in tables, the second that you could access any piece of data in a relational database by knowing its table name, key value, and finally, column name. This set forth the requirements that

- All columns must be atomic, that is, only a single value represented in a single column in a single row of a table.
- All rows of a table must be different.

In addition, to strengthen this stance, a First Normal Form was specified to require that an atomic value would not be extended to implement arrays, or, even worse, that position-based fields having multiple values would not be allowed. Hence, First Normal Form states that:

> *Every row should contain the same number of values, or in other words, no arrays, subtables, or repeating groups.*

This rule centers on making sure the implemented tables and columns are shaped properly for the relational languages that manipulate them (most importantly SQL). "Repeating groups" is an odd term that references having multiple values of the same type in a row, rather than splitting them into multiple rows. I will rephrase this "as rows in a table must contain the same number of values," rather than "repeating groups."

First Normal Form violations generally manifest themselves in the implemented model with data handling being far less optimal, usually because of having to decode multiple values stored where a single one should be or because of having duplicated rows that cannot be distinguished from one another. In this book, we are generally speaking of OLTP solutions, but even data warehousing databases generally follow First Normal Form in order to make queries work with the engine better. The later normal forms are less applicable because they are more concerned with redundancies in the data that make it harder to modify data, which is handled specially in data warehouse solutions.

All Columns Must Be Atomic

The goal of this requirement is that each column should represent only one value, not multiple values. This means there should be nothing like an array, no delimited lists, and no other types of multivalued columns that you could dream up represented by a single column. For example, consider a group of data like `1, 2, 3, 6, 7`. This likely represents five separate values. This group of data might not actually be multiple values as far as the design is concerned, but for darn sure it needs to be looked at.

One good way to think of atomicity is to consider whether you would ever need to deal with part of a column without the other parts of the data in that same column. In the list mentioned earlier—`1, 2, 3, 6, 7`—if the list is always treated as a single value in SQL, it might be acceptable to store the value in a single column.

www.it-ebooks.info

However, if you might need to deal with the value 3 individually, the value is definitely not in First Normal Form. It is also important to note that even if there is not a plan to use the list elements individually, you should consider whether it is still better to store each value individually to allow for future possible usage.

One variation on atomicity is for complex datatypes. Complex datatypes can contain more than one value, as long as

- There is always the same number of values.
- The values are rarely, if ever, dealt with individually.
- The values make up some atomic thing/attribute that could not be fully expressed with a single value.

For example, consider geographic location. Two values are generally used to locate something on Earth, these being the longitude and the latitude. Most of the time, either of these, considered individually, has some (if incomplete) meaning, but taken together, they pinpoint an exact position on Earth. Implementing as a complex type can give us some ease of implementing data-protection schemes and can make using the types in formulas easier.

When it comes to test of atomicity, the test of reasonability is left up to the designer (and other designers who inherit the work later, of course), but generally speaking, the goal is that any data you ever need to deal with as a single value is modeled as its own column, so it's stored in a column of its own (for example, as a search argument or a join criterion). As an example of taking atomicity to the extreme, consider a text document with ten paragraphs. A table to store the document might easily be implemented that would require ten different rows (one for each paragraph), but there's little reason to design like that, because you'll be unlikely to deal with a paragraph as a single value in the SQL database language. Of course, if your SQL is often counting the paragraphs in documents, that approach might just be the solution you are looking for (never let anyone judge your database without knowledge of the requirements!).

As further examples, consider some of the common locations where violations of this rule of First Normal Form often can be found:

- E-mail addresses
- Names
- Telephone numbers

Each of these gives us a slightly different kind of issue with atomicity that needs to be considered when designing columns.

E-mail Addresses

In an e-mail message, the e-mail address is typically stored in a format such as the following, using encoding characters to enable you to put multiple e-mail addresses in a single value:

```
name1@domain1.com;name2@domain2.com;name3@domain3.com
```

In the data storage tier of an e-mail engine, this is the optimum format. The e-mail address columns follow a common format that allows multiple values separated by semicolons. However, if you need to store the values in a relational database, storing the data in this format is going to end up being a problem, because it represents more than one e-mail address in a single `email` column and leads to difficult coding in the set oriented, multirow form that SQL is good for. In Chapter 10, it will become apparent from an internal engine standpoint why this is, but here, we will look at a practical example of how this will be bothersome when working with the data in SQL.

For example, if users are allowed to have more than one e-mail address, the value of an `email` column might look like this: `tay@bull.com; norma@liser.com`. Consider too that several users in the database might use the `tay@bull.com` e-mail address (for example, if it were the family's shared e-mail account).

www.it-ebooks.info

■ **Note** In this chapter, I will use the character = to underline the key columns in a table of data, and the – character for the nonkey attributes in order to make the representation easier to read without explanation.

Following is an example of some unnormalized data. In the following table, `PersonId` is the key column, while `FirstName` and `EmailAddresses` are the nonkey columns (not necessarily correct, of course as you will see.

```
PersonId  FirstName  EmailAddresses
========  ---------  ----------------------------------------------
0001003   Tay        tay@bull.com;taybull@hotmail.com;tbull@gmail.com
0003020   Norma      norma@liser.com
```

Consider the situation when one of the addresses changes. For example, we need to change all occurrences of `tay@bull.com` to `family@bull.com`. You could execute code such as the following to update every person who references the `tay@bull.com` address:

```
UPDATE dbo.person
SET EmailAddress=REPLACE(EmailAddresses,'tay@bull.com','family@bull.com')
WHERE ';'+emailAddress+';' LIKE '%;tay@bull.com;%';
```

This code might not seem like that much trouble to write and execute, but what about the case where there is also the e-mail address `bigtay@bull.com`? Our code invalidly replaces that value as well. How do you deal with that problem? Now, you have to start messing with adding semicolons to make sure the data fits just right for your search criteria so you don't get partial names. And that approach is fraught with all sorts of potential errors in making sure the format of the data stays the same. (For example, `"email;"@domain.com` is, in fact, a valid e-mail address based on the e-mail standard! See `http://en.wikipedia.org/wiki/E-mail_address#Valid_email_addresses`.) Data in the table should have meaning, not formatting. You format data for use, not for storage. It is easy to format data with the UI or even with SQL. Don't format data for storage.

Consider also if you need to count how many distinct e-mail addresses you have. With multiple e-mail addresses inline, using SQL to get this information is nearly impossible. But, you could implement the data correctly with each e-mail address represented individually in a separate row. Reformat the data as two tables, one for the `Person`:

```
PersonId FirstName
======== ---------
0001003  Tay
0003020  Norma
```

and a second table for a person's e-mail addresses:

```
PersonId EmailAddress
======== ===================
0001003  tay@bull.com
0001003  taybull@hotmail.com
0001003  tbull@gmail.com
0003020  norma@liser.com
```


www.it-ebooks.info

Now, an easy query determines how many e-mail addresses there are per person:

```
SELECT PersonId, COUNT(*) as EmailAddressCount
FROM dbo.PersonEmailAddress
GROUP BY PersonId;
```

Beyond being broken down into individual rows, e-mail addresses can be broken down into two or three obvious parts based on their format. A common way to break up these values is into the following parts:

- AccountName: name1
- Domain: domain1.com

Whether storing the data as multiple parts is desirable will usually come down to whether you intend to access the individual parts separately in your code. For example, if all you'll ever do is send e-mail, a single column (with a formatting constraint!) is perfectly acceptable. However, if you need to consider what domains you have e-mail addresses stored for, then it's a completely different matter.

Finally, a domain consists of two parts: domain1 and com. So you might end up with this:

```
PersonId Name    Domain  TopLevelDomain EmailAddress (calculated)
======== ======= ======= ============== ------------------------
0001003  tay     bull    com            tay@bull.com
0001003  taybull hotmail com            taybull@hotmail.com
0001003  tbull   gmail   com            tbull@gmail.com
0003020  norma   liser   com            norma@liser.com
```

At this point, you might be saying "What? Who would do that?" First off, I hope your user interface wouldn't force the users to enter their addresses one section at a time in either case, since parsing into multiple values is something the interface can do easily (and needs to do to at least somewhat to validate the format of the e-mail address.) Having the interface that is validating the e-mail addresses do the splitting is natural (as is having a calculated column to reconstitute the e-mail for normal usage).

The purpose of separating e-mail addresses into sections is another question. First off, you can start to be sure that all e-mail addresses are at least legally formatted. The second answer is that if you ever have to field questions like "what are the top ten services are our clients using for e-mail?" you can execute a query such as

```
SELECT TOP 10 Domain+'.'+TopLevelDomain AS Domain, COUNT(*) AS DomainCount
FROM PersonEmailAddress
GROUP BY Domain, TopLevelDomain
ORDER BY DomainCount;
```

Is this sort of data understanding necessary to your system? Perhaps and perhaps not. The point of this exercise is to help you understand that if you get the data broken down to the level in which you will query it, life will be easier, SQL will be easier to write, and your client will be happier?

Keep in mind, though, that you can name the column singularly and you can ask the user nicely to put in proper e-mail addresses, but if you don't protect the format, you will likely end up with your e-mail address table looking like this:

```
PersonId EmailAddress
======== ================================================
0001003  tay@bull.com
0001003  tay@bull.com;taybull@hotmail.com;tbull@gmail.com
0001003  tbull@gmail.com
0003020  norma@liser.com
```


www.it-ebooks.info

E-mail address values are unique in this example, but clearly do not represent single e-mail addresses. Every user represented in this data will now get lies as to how many addresses are in the system, and Tay is going to get duplicate e-mails, making your company look either desperate or stupid.

Names

Names are a fairly special case, as people in Western culture generally have three parts to their names. In a database, the name is often used in many ways: Simply first and last when greeting someone we don't know, first only when we want to sound cordial, and all three when we need to make our child realize we are actually serious.

Consider the name Rei Leigh Badezine. The first name, middle name, and last name could be stored in a single column and used. Using string parsing, you could get the first and last name if you needed it. Parsing seems simple, assuming every name is formatted just so. Add in names that have more complexity though, and parsing becomes a nightmare.

However, having multiple columns for holding names has its own set of struggles. A very common form that people tend to use is to just "keep it simple" and make one big column:

```
PersonId FullName
======== ----------------------
00202000 R. Lee Ermey
02300000 John Ratzenberger
03230021 Javier Fernandez Pena
```

This "one, big column" approach saves a lot of formatting issues for sure, but it has a lot of drawbacks as well. The problem with this approach is that is it is really hard to figure out what the first and last name are, because we have three different sorts of names formatted in the list. The best we could do is parse out the first and last parts of the names for reasonable searches (assuming no one has only one name!)

Consider you need to find the person with the name John Ratzenberger. This is easy:

```
SELECT FirstName, LastName
FROM Person
WHERE FullName = 'John Ratzenberger';
```

But what if you need to find anyone with a last name of Ratzenberger? This gets more complex, if not to code, certainly for the relational engine that works best with atomic values:

```
SELECT FirstName, LastName
FROM Person
WHERE FullName LIKE '%Ratzenberger';
```

Consider next the need of searching for someone with a middle name of Fernandez. This is where things get really muddy and very difficult to code correctly. So instead of just one big column, consider instead the following, more proper method of storing names. This time, each name-part gets its own column:

```
PersonId FirstName MiddleName LastName     FullName (calculated)
======== --------- ---------- --------     --------------------
00202000 R.        Lee        Ermey        R. Lee Ermey
02300000 John      NULL       Ratzenberger John Ratzenberger
03230021 Javier    Fernandez  Pena         Javier Fernandez Pena
```


www.it-ebooks.info

I included a calculated column that reconstitutes the name like it started and included the period after R. Lee Ermey's first name because it is an abbreviation. Names like his can be tricky, because you have to be careful as to whether or not this should be "R. Lee" as a first name or managed as two names. I would also advise you that, when creating interfaces to save names, it is almost always going to be better to provide the user with first, middle, and surname fields to fill out. Then allow the user to decide which arts of a name go into which of those columns. Leonardo Da Vinci is generally considered to have two names, not three. But Fred Da Bomb (who is an artist, just not up to Leonardo's quality), considers Da as his middle name. Allow your users to enter names as they see fit.

The prime value of doing more than having a blob of text for a name is in search performance. Instead of doing some wacky parsing on every usage and hoping everyone paid attention to the formatting, you can query by name using the following, simple, and easy to understand approach:

```
SELECT firstName, lastName
FROM person
WHERE firstName = 'John' AND lastName = 'Ratzenberger';
```

Not only does this code look a lot simpler than the code presented earlier, it works tremendously better. Because we are using the entire column value, indexing operations can be used to make searching easier. If there are only a few Johns in the database, or only a few Ratzenberger's (perhaps far more likely unless this is the database for the Ratzenberger family reunion), the optimizer can determine the best way to search.

Finally, the reality of a customer-oriented database may be that you need to store seemingly redundant information in the database to store different/customizable versions of the name, each manually created. For example, you might store versions of a person's name to be used in greeting the person (`GreetingName`), or to reflect the person likes to be addressed in correspondence (`UsedName`):

```
PersonId  FirstName  MiddleName  LastName     UsedName          GreetingName
========  ---------  ----------  -----------  ----------------  ------------
00202000  R.         Lee         Ermey        R. Lee Ermey      R. Lee
02300000  John       NULL        Ratzenberger John Ratzenberger John
03230021  Javier     Fernandez   Pena         Javier Pena       Javier
```

Is this approach a problem with respect normalization? Not at all. The name used to talk the person might be Q-dog, and the given name Leonard. Judging the usage is not our job; our job is to model it exactly the way it needs to be. The problem is that the approach definitely is a problem for the team to manage. In the normal case, if the first name changes, the used name and greeting name probably need to change and are certainly up for review. Minimally, this is the sort of documentation you will want to provide, along with user interface assistance, to our good friends the developers.

■ **Tip** Names are an extremely important part of a customer system. There is at least one hotel in Chicago I would hesitate to go back to because of what they called me in a very personal sounding thank you e-mail, and when I responded that it was wrong (in my most masculine-sounding typing), they did not reply.

Telephone Numbers

American telephone numbers are of the form 423-555-1212, plus some possible extension number. From our previous examples, you can see that several columns are probably in the telephone number value. However, complicating matters is that frequently the need exists to store more than just American telephone numbers in a database. The decision on how to handle this situation might be based on how often the users store

www.it-ebooks.info

international phone numbers, because it would be a hard task to build a table or set of tables to handle every possible phone format.

So, for an American- or Canadian-style telephone number, you can represent the standard phone number with three different columns for each of the following parts, AAA-EEE-NNNN:

- *AAA (Area code)*: Indicates a calling area located within a state
- *EEE (Exchange)*: Indicates a set of numbers within an area code
- *NNNN (Suffix*): Number used to make individual phone numbers unique

Here is where it gets tricky. If every phone number fits this format because you only permit calling to numbers in the North America, having three columns to represent each number would be a great solution. You might, too, want to include extension information. The problem is that all it takes is a single need to allow a phone number of a different format to make the pattern fail. So what do you do? Have a single column and just let anyone enter anything they want? That is the common solution, but you will all too frequently get users entering anything they want, and some stuff they will swear they didn't. Constrain values at the database level? That will make things better, but sometimes, you lose that battle because the errors you get back when you violate a CHECK constraint aren't very nice, and those people who enter the phone number in other reasonably valid formats get annoyed (of course, if the data is checked elsewhere, where does bad data come from?).

Why does it matter? Well, if a user misses a digit, you no longer will be able to call your customers to thank them or to tell them their products won't be on time. Plus, new area codes are showing up all of the time, and in some cases, phone companies split an area code and reassign certain exchanges to a new area code. The programming logic required to change part of a multipart value can be confusing. Take for example the following set of phone numbers:

```
PhoneNumber
============
615-555-4534
615-434-2333
```

The code to modify an existing area code to a new area code is pretty messy and certainly not the best performer. Usually, when an area code splits, it's for only certain exchanges. Assuming a well-maintained format of AAA-EEE-NNNN where AAA equals area code, EEE equals exchange, and NNNN equals the phone number, the code looks like this:

```
UPDATE dbo.PhoneNumber
SET PhoneNumber = '423' + substring(PhoneNumber,4,8)
WHERE substring(PhoneNumber,1,3) = '615'
 AND substring(PhoneNumber,5,3) IN ('232','323',...,'989');--area codes generally
                                                           --change for certain
                                                           --exchanges
```

This code requires perfect formatting of the phone number data to work, and unless the formatting is forced on the users, perfection is unlikely to be the case. Even a slight change, as in the following values

```
PhoneNumber
============
615-555-4534
615-434-2333
```

and you are not going to be able to deal with this data, because neither of these rows would be updated.

www.it-ebooks.info

Changing the area code is much easier if all values are stored in single, atomic containers, as shown here:

```
AreaCode  Exchange  PhoneNumber
========  ========  ===========
615       555       4534
615       434       2333
```

Now, updating the area code takes a single, easy-to-follow SQL statement, for example:

```
UPDATE dbo.PhoneNumber
SET AreaCode = '423'
WHERE AreaCode = '615'
 AND Exchange IN ('232','323',...,'989');
```

How you represent phone numbers is one of those case-by-case decisions. Using three separate values is easier for some reasons and, as a result, will be the better performer in almost all cases where you deal with only a single type of phone number. The one-value approach (with enforced formatting) has merit and will work, especially when you have to deal with multiple formats (be careful to have a key for what different formats mean and know that some countries have a variable number of digits in some positions).

You might even use a complex type to implement a phone number type. Sometimes, I use a single column with a check constraint to make sure all the dashes are in there, but I certainly prefer to have multiple columns unless the data isn't that important.

Dealing with multiple international telephone number formats complicates matters greatly, since only a few other countries use the same format as in the United States and Canada. And they all have the same sorts of telephone number concerns as we do with the massive proliferation of telephone number–oriented devices. Much like mailing addresses will be, how you model phone numbers is heavily influenced by how you will use them and especially how valuable they are to your organization. For example, a call center application might need far deeper control on the format of the numbers than would an application to provide simple phone functionality for an office. It might be legitimate to just leave it up to the user to fix numbers as they call them, rather than worry about programmability.

A solution that I have used is to have two sets of columns, with a column implemented as a calculated column that uses either the three-part number or the alternative number. Following is an example:

```
AreaCode  Exchange  PhoneNumber  AlternativePhoneNumber  FullPhoneNumber (calculated)
--------  --------  -----------  ----------------------  ============================
615       555       4534         NULL                    615-555-4534
615       434       2333         NULL                    615-434-2333
NULL      NULL      NULL         01100302030324          01100302030324
```

Then, I write a check constraint to make sure data follows one format or the other. This approach allows the interface to present the formatted phone number but provides an override as well. The fact is, with any shape of the data concerns you have, you have to make value calls on how important the data is and whether or not values that are naturally separate should actually be broken down in your actual storage. You could go much farther with your design and have a subclass for every possible phone number format on Earth, but this is likely overkill for most systems. Just be sure to consider how likely you are to have to do searches, like on the area code or partial phone numbers, and design accordingly.

www.it-ebooks.info

All Rows Must Contain the Same Number of Values

The First Normal Form says that every row in a table must have the same number of columns. There are two interpretations of this:

- Tables must have a fixed number of columns.
- Tables should be designed such that every row has a fixed number of values associated with it.

The first interpretation is simple and goes back to the nature of relational databases. You cannot have a table with a varying format with one row such as {Name, Address, Haircolor}, and another with a different set of columns such as {Name, Address, PhoneNumber, EyeColor}. This kind of implementation was common with record-based implementations but isn't possible with a relational database table. (Note that, internally, the storage of data in the storage engine does, in fact, look a lot like this because very little space is wasted in order to make better use of I/O channels and disk space. The goal of SQL is to make it easy to work with data and leave the hard part to other people).

The second is a more open interpretation. As an example, if you're building a table that stores a person's name and if one row has one name, all rows must have only one name. If they might have two, all rows must have precisely two (not one sometimes and certainly never three). If they may have a different number, it's inconvenient to deal with using SQL commands, which is the main reason a database is being built in an RDBMS! You must take some care with the concept of unknown values (NULLs) as well. The values aren't required, but there should always be the same number (even if the value isn't immediately known).

You can find an example of a violation of this rule of First Normal Form in tables that have several columns with the same base name suffixed (or prefixed) with a number, such as Payment1, Payment2, for example:

```
CustomerId  Name               Payment1   Payment2   Payment3
==========  -----------------  --------   --------   ----------
0000002323  Joe's Fish Market  100.03     NULL       120.23
0000230003  Fred's Cat Shop    200.23     NULL       NULL
```

Now, to add the next payment for Fred's, we might use some SQL code along these lines:

```
UPDATE dbo.Customer
SET Payment1 = CASE WHEN Payment1 IS NULL THEN 1000.00 ELSE Payment1 END
 Payment2 = CASE WHEN Payment1 IS NOT NULL AND Payment2 IS NULL THEN
 1000.00 ELSE Payment2 END,
 Payment3 = CASE WHEN Payment1 IS NOT NULL
  AND Payment2 IS NOT NULL
  AND Payment3 IS NULL
  THEN 1000.00 ELSE Paymen3 END
WHERE CustomerId = 0000230003;
```

And of course, if there were already three payments, you would not have made any changes at all. Obviously, a setup like this is far more optimized for manual modification, but our goal should be to eliminate places where people do manual tasks and get them back to doing what they do best, playing Solitaire . . . er, doing the actual business. Of course, even if the database is just used like a big spreadsheet, the preceding is not a great design. In the rare cases where there's always precisely the same number of values, then there's technically no violation of the definition of a table, or the First Normal Form. In that case, you could state a business rule that "each customer has exactly two payments." In my example though, what do you make of the fact that Payment2 is null, but Payment3 isn't? Did the customer skip a payment?

www.it-ebooks.info

Allowing multiple values of the same type in a single row still isn't generally a good design decision, because users change their minds frequently as to how many of whatever there are. For payments, if the person paid only half of their expected payment—or some craziness that people always seem to do—what would that mean?. To overcome these sorts of problems, you should create a child table to hold the values in the repeating payment columns. Following is an example. There are two tables. The first table holds customer names. The second table holds payments by customer.

```
CustomerId  Name
==========  -----------------
0000002323  Joe's Fish Market
0000230003  Fred's Cat Shop

CustomerId     Amount     Number    Date
==========     ------     ======    ----------
0000002323     100.03     1         2011-08-01
0000002323     120.23     3         2011-10-04
0000230003     200.23     1         2011-12-01
```

The first thing to notice is that I was able to add an additional column of information about each payment with relative ease—the date each payment was made. You could also make additions to the customer payment table to indicate if payment was late, whether additional charges were assessed, whether the amount is correct, whether the principal was applied, and more. Even better, this new design also allows us to have virtually unlimited payment cardinality, whereas the previous solution had a finite number (three, to be exact) of possible configurations. The fun part is designing the structures to meet requirements that are strict enough to constrain data to good values but loose enough to allow the user to innovate within reason.

Now, with payments each having their own row, adding a payment would be as simple as adding a new row to the `Payment` table:

```
INSERT dbo.Payment (CustomerId, Amount, Number, Date)
VALUES ('OOOOO2324', $300.00, 2, '2012-01-01');
```

You could calculate the payment number from previous payments, which is far easier to do using a set-based SQL statement, or the payment number could be based on something in the documentation accompanying the payment or even on when the payment is made. It really depends on your business rules. I would suggest however, if `paymentNumber` is simply a sequential value for display, I might not include it because it is easy to add a number to output using the `ROW_NUMBER()` windowing function, and maintaining such a number is usually more costly than calculating it.

Beyond allowing you to easily add data, designing row-wise clears up multiple annoyances, such as relating to the following tasks:

- Deleting a payment: Much like the update that had to determine what payment slot to fit the payment into, deleting anything other than the last payment requires shifting. For example, if you delete the payment in `Payment1`, then `Payment2` needs to be shifted to `Payment1`, `Payment3` to `Payment2`, and so on.
- Updating a payment: Say `Payment1` equals `10`, and `Payment2` equals `10`. Which one should you modify if you have to modify one of them because the amount was incorrect? Does it matter?

If your requirements really only allows three payments, it is easy enough to implement a constraint on cardinality. As I showed in Chapter 3 for data modeling, we control the number of allowable child rows using relationship cardinality. You can restrict the number of payments per customer using constraints or triggers

www.it-ebooks.info

(which will be described in more detail in Chapter 7) but whether or not you can implement something in the database is somewhat outside of this part of the database design process.

■ **Caution** Another common design uses columns such as `UserDefined1`, `UserDefined2`, . . . , `UserDefinedN` to allow users to store their own data that was not part of the original design. This practice is wickedly heinous for many reasons, one of them related to the proper application of First Normal Form. Second, using such column structures is directly against the essence of Codd's fourth rule involving a dynamic online catalog based on the relational model. That rule states that the database description is represented at the logical level in the same way as ordinary data so that authorized users can apply the same relational language to its interrogation that they apply to regular data.

Putting data into the database in more or less nameless columns requires extra knowledge about the system beyond what's included in the system catalogs (not to mention the fact that the user could use the columns for different reasons on different rows.) In Chapter 8, when I cover storing user-specified data (allowing user extensibility to your schema without changes in design), I will discuss more reasonable methods of giving users the ability to extend the schema at will.

All Rows Must Be Different

One of the most important things you must take care of when building a database is to make sure to have keys on tables to be able to tell rows apart. Although having a completely meaningful key isn't reasonable 100 percent of the time, it usually is very possible. An example is a situation where you cannot tell the physical items apart, such as perhaps a small can of corn (or a large one, for that matter). Two cans cannot be told apart, so you might assign a value that has no meaning as part of the key, along with the things that differentiate the can from other similar objects, such as large cans of corn or small cans of spinach. You might also consider keeping just a count of the objects in a single row, depending on your needs (which will be dictated by your requirements).

Often, designers are keen to just add an artificial key value to their table, using a GUID or an integer, but as discussed in Chapter 1, adding an artificial key might technically make the table comply with the letter of the rule, but it certainly won't comply with the purpose. The purpose is that no two rows represent the same thing. An artificial key by definition has no meaning, so it won't fix the problem. You could have two rows that represent the same thing because every meaningful value has the same value, with the only difference between rows being a system-generated value. Note that I am not against using an artificial key, just that it should rarely be the only defined key. As mentioned in Chapter 1, another term for such a key is a *surrogate key*, so named because it is a surrogate (or a stand in) for the real key.

A common approach is to use a date and time value to differentiate between rows. If the date and time value is part of the row's logical identification, such as a calendar entry or a row that's recording/logging some event, this is not only acceptable but ideal. Conversely, simply tossing on a date and time value to force uniqueness is no better than just adding a random number or GUID on the row.

As an example of how generated values lead to confusion, consider the following subset of a table with school mascots:

```
MascotId  Name
========  ------
1         Smokey
112       Smokey
4567      Smokey
979796    Smokey
```


www.it-ebooks.info

Taken as it is presented, there is no obvious clue as to which of these rows represent the real Smokey, or if there needs to be more than one Smokey, or if the data entry person just goofed up. It could be that the school name ought to be included to produce a key, or perhaps names are supposed to be unique, or even that this table should represent the people who play the role of each mascot. It is the architect's job to make sure that the meaning is clear and that keys are enforced to prevent, or at least discourage, alternative interpretations.

Of course, the reality of life is that users will do what they must to get their job done. Take, for example, the following table of data that represents books:

```
BookISBN  BookTitle     PublisherName Author
========= ------------- ------------- -------
111111111 Normalization Apress        Louis
222222222 T-SQL         Apress        Michael
```

The users go about life, entering data as needed. But when users realize that more than one author needs to be added per book, they will figure something out. What a user might figure out might look as follows:

```
444444444       DMV Book Simple Talk Tim
444444444-1     DMV Book Simple Talk Louis
```

The user has done what was needed to get by, and assuming the domain of `BookISBN` allows it, the approach works with no errors. However, *DMV Book* looks like two books with the same title. Now, your support programmers are going to have to deal with the fact that your data doesn't mean what you think it does. Ideally, at this point, you would realize that you need to add a table for authors, and you have the solution that will give you the results you want:

```
BookISBN         BookTitle           PublisherName
=========        -------------       -------------
111111111        Normalization       Apress
222222222        T-SQL               Apress
333333333        Indexing            Microsoft
444444444        DMV Book            Simple Talk

BookISBN        Author
=========       ========
111111111       Louis
222222222       Michael
333333333       Kim
444444444       Tim
444444444       Louis
```

Now, if you need information about an author's relationship to a book (chapters written, pay rate, etc.) you can add columns to the second table without harming the current users of the system. Yes, you end up having more tables, and yes, you have to do more coding up front, but if you get the design right, it just plain works. The likelihood of discovering all data cases such as this case with multiple authors to a book before you have "completed" your design, is fairly low, so don't immediately think it is your fault. Requirements are often presented that are not fully thought through such as claiming only one author per book. Sometimes it isn't that the requirements were faulty so much as the fact that requirements change over time. In this example, it

www.it-ebooks.info

could have been that during the initial design phase the reality at the time was that the system was to support only a single author. Ever changing realities are what makes software design such a delightful task as times. **

■ **Caution** Key choice is one of the most important parts of your database design. Duplicated data causes tremendous and obvious issues to anyone who deals with it. It is particularly bad when you do not realize you have the problem until it is too late.

Clues That an Existing Design Is Not in First Normal Form

When you are looking at database design to evaluate it, you can look quickly at a few basic things to see whether the data is in First Normal Form. In this section, we'll look at some of these ways to recognize whether data in a given database is already likely to be in First Normal Form. None of these clues is, by any means, a perfect test. Generally speaking, they're only clues that you can look for in data structures for places to dig deeper. Normalization is a moderately fluid set of rules somewhat based on the content and use of your data.

The following sections describe a couple of data characteristics that suggest that the data isn't in First Normal Form:

- String data containing separator-type characters
- Column names with numbers at the end
- Tables with no or poorly defined keys

This is not an exhaustive list, of course, but these are a few places to start.

String Data Containing Separator-Type Characters

Separator-type characters include commas, brackets, parentheses, semicolons, and pipe characters. These act as warning signs that the data is likely a multivalued column. Obviously, these same characters are often used in normal prose, so you need not go too far. For instance, if you're designing a solution to hold a block of text, you've probably normalized too much if you have a word table, a sentence table, and a paragraph table (if you had been considering it, give yourself three points for thinking ahead, but don't go there). In essence, this clue is basically aligned to tables that have structured, delimited lists stored in a single column rather than broken out into multiple rows.

Column Names with Numbers at the End

As noted, an obvious example would be finding tables with `Child1`, `Child2`, and similar columns, or my favorite, `UserDefined1`, `UserDefined2`, and so on. These kinds of tables are usually messy to deal with and should be considered for a new, related table. They don't have to be wrong; for example, your table might need exactly two values to always exist. In that case, it's perfectly allowable to have the numbered columns, but be careful that what's thought of as "always" is actually always. Too often, exceptions cause this solution to fail. "A person always has two forms of identification noted in fields `ID1` and `ID2`, except when . . ." In this case, "always" doesn't mean always.

These kinds of column are a common holdover from the days of flat-file databases. Multitable/multirow data access was costly, so developers put many fields in a single file structure. Doing this in a relational database system is a waste of the power of the relational programming language.

`Coordinate1` and `Coordinate2` might be acceptable in cases that always require two coordinates to find a point in a two-dimensional space, never any more or never any less (though `CoordinateX` and `CoordinateY` would likely be better column names).

www.it-ebooks.info

Tables with No or Poorly-Defined Keys

As noted in the early chapters several times, key choice is very important. Almost every database will be implemented with some primary key (though this is not a given in many cases). However, all too often the key will simply be a GUID or an identity-based value.

It might seem like I am picking on identity and GUIDs, and for good reason: I am. While I will almost always suggest you use surrogate keys in your implementations, you must be careful when using them as primary keys, and too often people use them incorrectly. Keeping row values unique is a big part of First Normal Form compliance and is something that should be high on your list of important activities.

Relationships Between Columns

The next set of normal forms to look at is concerned with the relationships between attributes in a table and, most important, the key(s) of that table. These normal forms deal with minimizing functional dependencies between the attributes. As discussed in Chapter 1, being functionally dependent implies that when running a function on one value (call it `Value1`), if the output of this function is *always* the same value (call it `Value2`), then `Value2` is functionally dependent on `Value1`.

As I mention, there are three normal forms that specifically are concerned with the relationships between attributes. They are

- Second Normal Form: Each column must be a fact describing the entire primary key (and the table is in First Normal Form)
- Third Normal Form: Non-primary-key columns cannot describe other non-primary-key columns. (and the table is in Second Normal Form)
- Boyce-Codd Normal Form (BCNF): All columns are fully dependent on a key. Every determinant is a key (and the table is in First Normal Form, not Second or Third since BCNF itself is a more strongly stated version that encompasses them both).

I am going to focus on the BCNF because it encompasses the other forms, and it is the most clear and makes the most sense based on today's typical database design patterns (specifically the use of surrogate keys and natural keys).

BCNF Defined

BCNF is named after Ray Boyce, one of the creators of SQL, and Edgar Codd, whom I introduced in the first chapter as the father of relational databases. It's a better-constructed replacement for both the Second and Third Normal Forms, and it takes the meanings of the Second and Third Normal Forms and restates them in a more general way. The BCNF is defined as follows:

- The table is already in First Normal Form.
- All columns are fully dependent on a key.
- A table is in BCNF if every determinant is a key.

Note that, to be in BCNF, you don't specifically need to be concerned with Second Normal Form or Third Normal Form. BCNF encompasses them both and changed the definition from the "primary key" to simply all defined keys. With today's almost universal de facto standard of using a surrogate key as the primary key, BCNF is a far better definition of how a properly designed database should be structured. It is my opinion that *most* of the time when someone says "Third Normal Form" that they are referencing something closer to BCNF.

www.it-ebooks.info

An important part of the definition of BCNF is that "every determinant is a key." I introduced determinants back in Chapter 1, but as a quick review, consider the following table of rows, with X defined as the key:

```
X   Y   Z
=== --- ---
1   1   2
2   2   4
3   2   4
```

X is unique, and given the value of X, you can determine the value of Y and Z. X is the determinant for all of the rows in the table. Now, given a value of Y, you can't determine the value of X, but you seemingly can determine the value of Z. When `Y = 1: Z = 2` and when `Y = 2: Z = 4`. Now before you pass judgment, this could be a coincidence. It is very much up to the requirements to help us decide if this determinism is incorrect. If the values of Z were arrived at by a function of Y*2, then Y would determine Z and really wouldn't need to be stored (eliminating user editable columns that are functional dependent on one another is one of the great uses of the calculated column in your SQL tables, and they manage these sorts of relationships).

When a table is in BCNF, any update to a nonkey column requires updating one and only one value. If Z is defined as Y*2, updating the Y column would require updating the Z column as well. If Y could be a key, this would be acceptable as well, but Y is not unique in the table. By discovering that Y is the determinant of Z, you have discovered that YZ should be its own independent table. So instead of the single table we had before, we have two tables that express the previous table with no invalid functional dependencies, like this:

```
X   Y
=== ---
1   1
2   2
3   2

Y   Z
=== ---
1   2
2   4
```

For a somewhat less abstract example, consider the following set of data, representing book information:

```
BookISBN  BookTitle      PublisherName    PublisherLocation
========= ------------- --------------   -----------------
111111111 Normalization  Apress           California
222222222 T-SQL          Apress           California
444444444 DMV Book       Simple Talk      England
```

`BookISBN` is the defined key, so every one of the columns should be completely dependent on this value. The title of the book is dependent on the book ISBN, and the publisher too. The concern in this table is the `PublisherLocation`. A book doesn't have a publisher location, a publisher does. So if you needed to change the publisher, you would also need to change the publisher location.

www.it-ebooks.info

To correct this situation, you need to create a separate table for publisher. Following is one approach you can take:

```
BookISBN      BookTitle        PublisherName
=========     -------------    -------------
111111111     Normalization    Apress
222222222     T-SQL            Apress
444444444     DMV Book         Simple Talk

Publisher       PublisherLocation
===========     -----------------
Apress          California
Simple Talk     England
```

Now, a change of publisher for a book requires only changing the publisher value in the `Book` table, and a change to publisher location requires only a single update to the `Publisher` table.

Partial Key Dependency

In the original definitions of the normal forms, we had second normal form that dealt with partial key dependencies. In BCNF, this is still a concern when you have defined composite keys (with more than one column making up the key). Most of the cases where you see a partial key dependency in an example are pretty contrived (and I will certainly not break than trend). Partial key dependencies deal with the case where you have a multicolumn key and, in turn, columns in the table that reference only part of the key.

As an example, consider a car rental database and the need to record driver information and type of cars the person will drive. Someone might (not you, certainly!) create the following:

```
Driver VehicleStyle Height EyeColor ExampleModel DesiredModelLevel
====== ============ ====== ======== ============ -----------------
Louis  CUV          6'0"   Blue     Edge         Premium
Louis  Sedan        6'0"   Blue     Fusion       Standard
Ted    Coupe        5'8"   Brown    Camaro       Performance
```

The key of driver plus vehicle style means that all of the columns of the table should reference both of these values. Consider the following columns:

- `Height`: Unless this is the height of the car, this references the driver and not car style.
- `EyeColor`: Clearly, this references the driver only, unless we rent Pixar car models.
- `ExampleModel`: This references the `VehicleStyle`, providing a model for the person to reference so they will know approximately what they are getting.
- `DesiredModelLevel`: This represents the model of vehicle that the driver wants to get. This is the only column that should be in this table.

www.it-ebooks.info

To transform the initial one table into a proper design, we will need to split the one table into three. The first one defines the driver and just has the driver's physical characteristics:

```
Driver  Height  EyeColor
======  ------  --------
Louis   6'0"    Blue
Ted     5'8"    Brown
```

The second one defines the car styles and model levels the driver desires:

```
Driver  VehicleStyle     DesiredModelLevel
======  ============     -----------------
Louis   CUV              Premium
Louis   Sedan            Standard
Ted     Coupe            Performance
```

And finally, we need one to define the types of car styles available and to give the example model of the car:

```
VehicleStyle    ExampleModel
============    ------------
CUV             Edge
Sedan           Fusion
Coupe           Camaro
```

Note that, since the driver was repeated multiple times in the original poorly designed sample data, I ended up with only two rows for the driver as the `Louis` entry's data was repeated twice. It might seem like I have just made three shards of a whole, without saving much space, and ultimately will need costly joins to put everything back together. The reality is that in a real database, the driver table would be large; the table assigning drivers to car styles is very thin (has a small number of columns), and the car style table will be very small. The savings from not repeating so much data will more than overcome the overhead of doing joins on reasonably designed tables. For darn sure, the integrity of the data is much more likely to remain at a high level because every single update will only need to occur in a single place.

Entire Key Dependency

Third Normal Form and BCNF deal with the case where all columns need to be dependent on the entire key. (Third Normal Form specifically dealt with the primary key, but BCNF expanded it to include all defined keys.) When we have completed our design, and it is meets the standards of BCNF, every possible key is defined.

In our previous example we ended up with a driver table. That same developer, when we stopped watching, made some additions to the table to get an idea of what the driver currently drives:

```
Driver    Height    EyeColor    Vehicle Owned      VehicleDoorCount     VehicleWheelCount
======    ------    --------    ---------------    ----------------     -----------------
Louis     6'0"      Blue        Hatchback          3                    4
Ted       5'8"      Brown       Coupe              2                    4
Rob       6'8"      NULL        Tractor trailer    2                    18
```


www.it-ebooks.info

To our trained eye, it is pretty clear almost immediately that the vehicle columns aren't quite right, but what to do? You could make a row for each vehicle, or each vehicle type, depending on how specific the need is for the usage. Since we are basically trying to gather demographic information about the user, I am going to choose vehicle type to keep things simple (and since it is a great segue to the next section). The vehicle type now gets a table of its own, and we remove from the driver table all of the vehicle pieces of information and create a key that is a coagulation of the values for the data in row:

```
VehicleTypeId       VehicleType          DoorCount      WheelCount
==============      ---------------      ---------      -----------
3DoorHatchback      Hatchback            3              4
2DoorCoupe          Coupe                2              4
TractorTrailer      Tractor trailer      2              18
```

And the driver table now references the vehicle type table using its key:

```
Driver  VehicleTypeId   Height  EyeColor
======  --------------  ------  --------
Louis   3DoorHatchback  6'0"    Blue
Ted     2DoorCoupe      5'8"    Brown
Rob     TractorTrailer  6'8"    NULL
```

Note that for the vehicle type table in this model, I chose to implement a smart surrogate key for simplicity, and because it is a common method that people use. A short code is concocted to give the user a bit of readability, then the additional columns are there to use in queries, particularly when you need to group or filter on some value (like if you wanted to send an offer to all drivers of three-door cars to drive a luxury car for the weekend!) It has drawbacks that are the same as the normal form we are working on if you aren't careful (the smart key has redundant data in it), so using a smart key like this is a bit dangerous. But what if we decided to use the natural key? You would end up with two tables that look like this:

```
Driver     Height    EyeColor    Vehicle Owned       VehicleDoorCount
=======    ------    --------    ---------------     ----------------
Louis      6'0"      Blue        Hatchback           3
Ted        5'8"      Brown       Coupe               2
Rob        6'8"      NULL        Tractor trailer     2

VehicleType           DoorCount       WheelCount
===============       =========       ----------
Hatchback             3               4
Coupe                 2               4
Tractor trailer       2               18
```

The driver table now has almost the same columns as it had before (less the `WheelCount` which does not differ from a three- or five-door hatchback, for example), referencing the existing tables columns, but it is a far more flexible solution. If you want to include additional information about given vehicle types (like towing capacity, for example), you could do it in one location and not in every single row, and users entering driver information can only use data from a given domain that is defined by the vehicle type table. Note too that the two

www.it-ebooks.info

solutions proffered are semantically equivalent in meaning but have two different solution implementations that will have an effect on implementation, but not the meaning of the data in actual usage.

Surrogate Keys Effect on Dependency

When you use a surrogate key, it is used as a stand-in for the existing key. To our previous example of driver and vehicle type, let's make one additional example table set, using a meaningless surrogate value for the vehicle type key, knowing that the natural key of the vehicle type set is `VehicleType` and `DoorCount`:

```
Driver      VehicleTypeId     Height      EyeColor
======      -------------     ------      --------
Louis       1                 6'0"        Blue
Ted         2                 5'8"        Brown
Rob         3                 6'8"        NULL

VehicleTypeId     VehicleType          DoorCount      WheelCount
=============     ------------         ---------      ----------
1                 Hatchback            3              4
2                 Coupe                2              4
3                 Tractor trailer      2              18
```

I am going to cover key choices a bit more in Chapter 6 when I discuss uniqueness patterns, but suffice it to say that, for design and normalization purposes, using surrogates doesn't change anything except the amount of work it takes to validate the model. Everywhere the `VehicleTypeId` of 1 is referenced, it is taking the place of the natural key of `VehicleType`, `DoorCount`; and you must take this into consideration. The benefits of surrogates are more for programming convenience and performance but they do not take onus away from you as a designer to expand them for normalization purposes.

For an additional example involving surrogate keys, consider the case of an employee database where you have to record the driver's license of the person. We normalize the table and create a table for driver's license and we end up with the model snippet in Figure 5-1. Now, as you are figuring out whether or not the

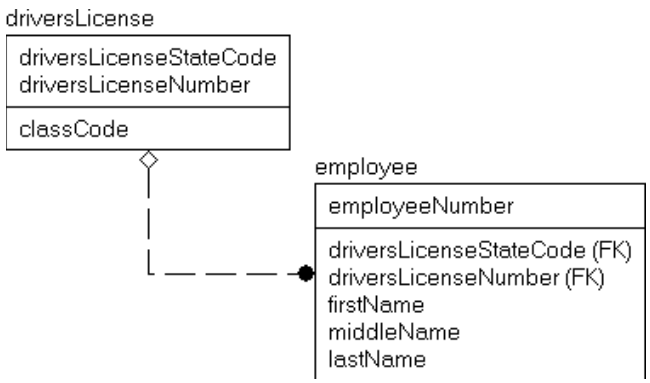


Figure 5-1. *driversLicense and employee tables with natural keys*

www.it-ebooks.info

employee table is in proper BCNF, you check out the columns and you come to `driversLicenseStateCode` and `driversLicenseNumber`. Does an employee have a ? Not exactly, but a drivers license does. When columns are part of a foreign key, you have to consider the entire foreign key as a whole. So can an employee have a driver's license? Of course.

What about using surrogate keys? Well this is where the practice comes with additional cautions. In Figure 5-2, I have remodeled the table using surrogate keys for each of the tables.

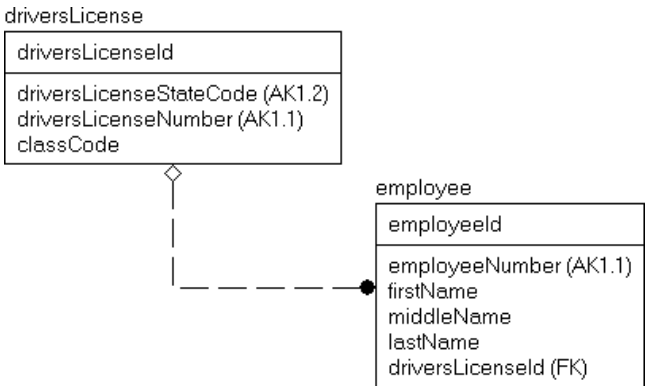


Figure 5-2. *driversLicense and employee tables with surrogate keys*

This design looks cleaner in some ways, and with the very well named columns from the natural key, it is a bit easier to see the relationships in these tables, and it is not always possible to name the various parts of the natural key as clearly as I did. In fact, the state code likely would have a domain of its own and might be named `StateCode`. The major con is that there is a hiding of implementation details that can lead to insidious multitable denormalizations. For example, take the following addition to the model made by designers who weren't wearing their name-brand thinking caps:

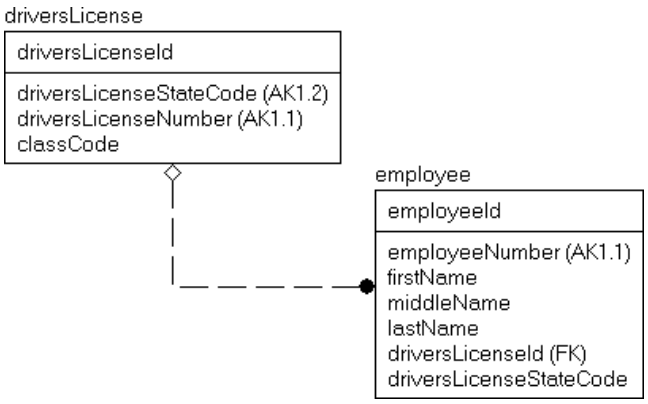


Figure 5-3. *driversLicense and employee tables with improper normalization*

www.it-ebooks.info

The users wanted to know the state code from the driver's license for the employer, so they added it do the employee table because it wasn't easily visible in the table. Now in essence, here is what we have in the employee table once we expand the columns from the natural key of the `driversLicense` table:

- `employeeNumber`
- `firstName`
- `middleName`
- `lastName`
- `driversLicenseStateCode (driversLicense)`
- `driversLicenseNumber (driversLicense)`
- `driversLicenseStateCode`

The state code is duplicated just to save a join to the table where it existed naturally, so the fact that we are using surrogate keys to simplify some programming tasks is not complicated by the designer's lack of knowledge (or possibly care) for why we use surrogates.

While the `driversLicense` example is a simplistic case that only a nonreader of this book would perpetrate, in a real model, the parent table could be five or six levels away from the child, with all tables using single key surrogates, hiding lots of natural relationships. It looks initially like the relationships are simple one-table relationships, but a surrogate key takes the place of the natural key, so in a model like in Figure 5-4, the keys are actually more complex than it appears.

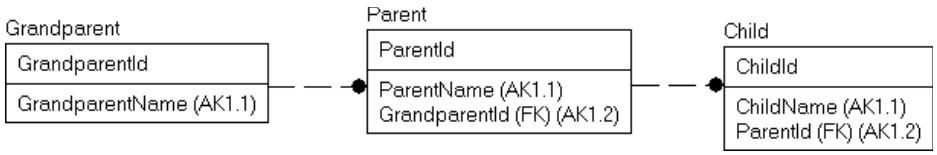


Figure 5-4. *Tables chained to show key migration*

The full key for the `Grandparent` table is obvious, but the key of the `Parent` table is a little bit less so. Where you see the surrogate key of `GrandparentId` in the `Parent`, you need to replace it with a natural key from `Grandparent`. So the key to `Parent` is `ParentName, GrandParentName`. Then with child, it is the same thing, so the key becomes `ChildName, ParentName, GrandparentName`. This is the key you need to compare your other attributes against to make sure that it is correct.

■ **Note** A lot of purists really hate surrogates because of how much they hide the interdependencies, and I would avoid them if you are unwilling to take the time to understand (and document) the models you are creating using them. As a database design book reader, unless you stumbled on this page looking for pictures of fashion models, I am going to assume this won't be an issue for you.

---

www.it-ebooks.info

Dependency Between Rows

A consideration when discussing BCNF is data that is dependent on data in a different row, possibly even in a different table. A common example of this is summary data. It is an epidemic among row-by-row thinking programmers who figure that it is very costly to calculate values using set-based queries. So say, you have objects for invoice and invoice line items like the following tables of data, the first being an invoice, and the second being the line items of the invoice:

```
InvoiceNumber   InvoiceDate     InvoiceAmount
=============   -----------     -------------
000000000323    2011-12-23      100

InvoiceNumber   ItemNumber   InvoiceDate   Product   Quantity   ProductPrice   OrderItemId
=============   ==========   -----------   -------   --------   ------------   -----------
000000000323    1            2011-12-23    KL7R2     10         8.00           1232322
000000000323    2            2011-12-23    RTCL3     10         2.00           1232323
```

There are two issues with this data arrangement.

- `InvoiceAmount` is just the calculated value of `SUM(Quantity * ProductPrice)`.
- `InvoiceDate` in the line item is just repeated data from `Invoice`.

Now, your design has become a lot more brittle, because if the invoice date changes, you will have to change all of the line items. The same is likely true for the `InvoiceAmount` value as well. However, it may not be so. You have to question whether or not the `InvoiceAmount` is a true calculation. In many cases, while the amounts actually seem like they are calculated, they may be there as a check. The value of 100 may be manually set as a check to make sure that no items are changed on the line items. The important part of this topic is that you must make sure that you store all data that is independent, regardless of how it might appear. When you hit on such natural relationships that are not implemented as dependencies, some form of external controls must be implemented, which I will talk about more in the later section titled "Denormalization".

There is one other possible problem with the data set, and that is the `ProductPrice` column. The question you have to consider is the life and modifiability of the data. At the instant of creating the order, the amount of the product is fetched and normally used as the price of the item. Of course, sometimes you might discount the price, or just flat change it for a good customer (or noisy one!), not to mention that prices can change. Of course, you could use the price on the order item row that is referenced. But you still might want to make changes here if need be.

The point of this work is that you have to normalize to what is actually needed, not what seems like a good idea when doing design. This kind of touch will be where you spend a good deal of time in your designs. Making sure that data that is editable is actually stored, and data that can be retrieved from a different source is not stored.

Clues That Your Database Is Not in BCNF

In the following sections, I will present a few of the flashing red lights that can tell you that your design isn't in BCNF.

- Multiple columns with the same prefix
- Repeating groups of data
- Summary data

www.it-ebooks.info

Of course, these are only the truly obvious issues with tables, but they are very representative of the types of problems that you will frequently see in designs that haven't been done well (you know, by those people who haven't read this book!)

Multiple Columns with the Same Prefix

The situation of repeating key column prefixes is one of the dead giveaways. Going back to our earlier example table

```
BookISBN    BookTitle       PublisherName    PublisherLocation
=========   -------------   -------------    -----------------
111111111   Normalization   Apress           California
222222222   T-SQL           Apress           California
444444444   DMV Book        Simple Talk      England
```

the problem identified was in the `PublisherLocation` column that is functionally dependent on `PublisherName`. Prefixes like the "Publisher" in these two column names are a rather common tip-off, especially when designing new systems. Of course, having such an obvious prefix on columns such as `Publisher%` is awfully convenient, but it isn't always the case in real-life examples that weren't conjured up as an illustration.

Sometimes, rather than having a single table issue, you find that the same sort of information is strewn about the database, over multiple columns in multiple tables. For example, consider the tables in Figure 5-5.

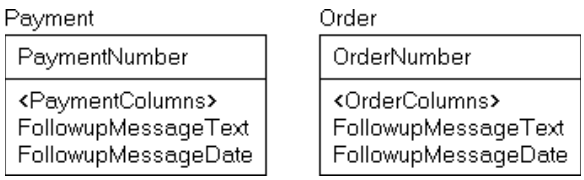


Figure 5-5. *Payment and Order with errant Followup columns*

The tables in Figure 5-5 are a glowing example of information that is being wasted by not having it consolidated in the same table. Most likely, you want to be reasonable with the amount of messages you send to your customers. Send too few and they forget you, too many and they get annoyed by you. By consolidating the data into a single table, it is far easier to manage. Figure 5-6 shows a better version of the design.

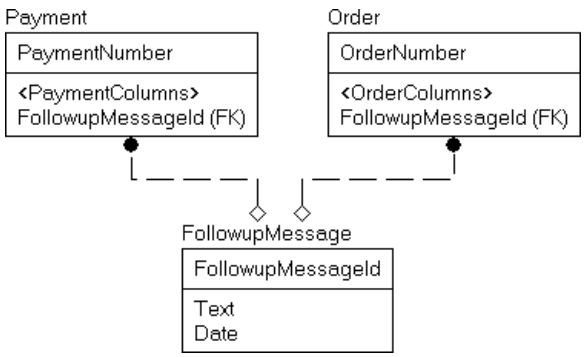


Figure 5-6. *Payment and Order with added Followup object*

www.it-ebooks.info

Repeating Groups of Data

More difficult to recognize are the repeating groups of data. Imagine executing multiple SELECT statements on a table, each time retrieving all rows (if possible), ordered by each of the important columns. If there's a functionally dependent column, you'll see that in form of the dependent column taking on the same value Y for a given column value X.

Take a look at some example entries for the tables we just used in previous sections:

```
BookISBN    BookTitle       PublisherName    PublisherLocation
=========   -------------   -------------    -----------------
111111111   Normalization   Apress           California
222222222   T-SQL           Apress           California
444444444   DMV Book        Simple Talk      England
```

The repeating values (Apress and California) are a clear example of something that is likely amiss. It isn't a guarantee, of course, but you can look for data such as this by careful queries. In essence, you can profile your data to identify suspicious correlations that deserve a closer look. Sometimes, even if the names are not so clearly obvious, finding ranges of data such as in the preceding example can be very valuable.

Summary Data

One of the most common violations of BCNF that might not seem obvious is *summary data*. This is where columns are added to the parent table that refer to the child rows and summarize them. Summary data has been one of the most frequently necessary evils that we've had to deal with throughout the history of the relational database server. There might be cases where calculated data needs to be stored in a table in violation of Third Normal Form, but in logical modeling, there's *absolutely* no place for it. Not only is summary data not functionally dependent on nonkey columns, it's dependent on columns from a different table altogether. This causes all sorts of confusion, as I'll demonstrate. Summary data should be reserved either for physical design or for implementation in reporting/data warehousing databases.

Take the example of an auto dealer, as shown in Figure 5-7. The dealer system has a table that represents all the types of automobiles it sells, and it has a table recording each automobile sale.

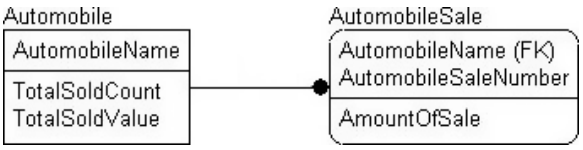


Figure 5-7. *The auto dealer submodel*

Summary data generally has no part in the logical model you will be creating, because the sales data is available in another table. Instead of accepting that the total number of vehicles sold and their value is available, the designer has decided to add columns in the parent table that refer to the child rows and summarize them.

Is this required for the implementation? It's unlikely, but possible, depending on performance needs. (It could be that the total values are used millions of times a day, with very infrequent changes to the data that makes up the total.) However, it's common that the complexity of the implemented system has most likely

www.it-ebooks.info

increased by an order of magnitude, because we'll have to have triggers on the AutomobileSale table that calculate these values for any change in the AutomobileSale table. If this is a highly active database with frequent rows added to the AutomobileSale table, this tends to slow the database down considerably. On the other hand, if it's an often inactive database, then there will be few rows in the child table, so the performance gains made by quickly being able to find the numbers of vehicles sold and their value will be small anyway.

The point is that in logical modeling, including summary data on the model isn't desirable, because the data modeled in the total column exists in the Sales table. What you are actually modeling is usage, not the structure of the data. Data that we identify in our logical models should be modeled to exist in only one place, and any values that could be calculated from other values shouldn't be represented on the model. This aids in keeping the integrity of the design of the data at its highest level possible.

■ **Tip** One way of dealing with summary data is to use a view. An automobile view might summarize the automobile sales. In some cases, you can index the view, and the data is automatically maintained for you. The summarized data is easier to maintain using the indexed view, though it can have negative performance repercussions on modifications but positive ones on reads. Only testing your actual situation will tell, but this is not the implementation part of the book! I'll discuss indexes in some detail in Chapter 10.

Positional Meaning

The last point I want to make about BCNF type issues is that you must be truly careful about the meaning of the data you are normalizing, because as you get closer and closer to the goal of one table having one meaning, almost every column will have one place where it makes sense. For example, consider the following table of data:

```
CustomerId      Name      EmailAddress1       EmailAddress2       AllowMarketingByEmailFlag
==========      -----     ---------------     ---------------     -------------------------
A0000000032     Fred      fred@email.com      fred2@email.com     1
A0000000033     Sally     sally@email.com     NULL                0
```

To get this table into First Normal Form, you should immediately recognize that we need to implement a table to hold the e-mail address for the customer. The questionable attribute is the AllowMarketingByEmailFlag, which denotes whether or not we wish to market to this customer by e-mail. Is this an attribute about the e-mail address? Or the customer?

Without additional knowledge from the client, it must be assumed that the AllowMarketingByEmailFlag column applies to how we will market to the customer, so it should remain on the customer table like this:

```
CustomerId      Name    AllowMarketingByEmailFlag
==========      -----   -------------------------
A0000000032     Fred    1
A0000000033     Sally   0

CustomerId      EmailAddress        EmailAddressNumber
==========      ---------------     ==================
A0000000032     fred@email.com      1
A0000000032     fred2@email.com     2
A0000000033     sally@email.com     1
```


www.it-ebooks.info

You will also notice that I made the key of the customer e-mail address table `CustomerId`, `EmailAddressNumber` and not `EmailAddress`. Without further knowledge of the system, it would be impossible to know if it was acceptable to have duplication in the two columns. It really boils down to the original purpose of having multiple `EmailAddress` values, and you have to be careful about what the customers may have been using the values for. In a situation I recently was working on, half of the users used the latter addresses as history of old e-mail addresses and the other half as a backup e-mail for contacting the customer. For the history e-mail address values, it certainly could make sense to add start and end date values to tell when and if the address is still valid, particularly for customer relationship management systems. But at the same time, it could make sense to have only current customer information in your OLTP system and move history of to an archival or data warehouse database instead.

Finally, consider the following scenario. A client sells an electronic product that is delivered by e-mail. Sometimes, it can take weeks before the order is fulfilled and shipped. So the designer of the system created the following three table solution (less the sales order line item information about the product that was ordered):

```
CustomerId      Name    AllowMarketingByEmailFlag
==========      ----    -------------------------
A0000000032     Fred    1

CustomerId      EmailAddress         EmailAddressNumber
==========      ---------------      ==================
A0000000032     fred@email.com       1
A0000000032     fred2@email.com      2

SalesOrderId  OrderDate    ShipDate     CustomerId   EmailAddressNumber   ShippedToEmailAddress
============  ----------   ----------   ----------   ------------------   ---------------------
1000000242    2012-01-01   2012-01-02   A0000000032  1                    fred@email.com
```

What do you figure the purpose is of the redundant e-mail address information? Is it a normalization issue? No, because although the `ShippedToEmailAddress` may be exactly the same as the e-mail address for the e-mail address table row with the related e-mail address number, what if the customer changed e-mail addresses and then called in to ask where you shipped the product? If you only maintained the link to the customer's current e-mail address, you wouldn't be able to know what the e-mail address was when it shipped.

The point of this section has been to think before you eliminate what seems like redundant data. Good naming standards, such as spelling out `ShippedToEmailAddress` during the logical database design phase, are a definite help to make sure other developers/architects know what you have in mind for a column that you create.

Tables with Multiple Meanings

So, assuming you (A) have done some work with databases before getting this deep in my book and (B) haven't been completely self-taught while living underneath 100,000 pounds of granite, you probably are wondering why this chapter on normalization didn't end a few pages back. You probably have heard that Third Normal Form (and assuming you are paying attention, BCNF) is far enough. That is often true, but not because the higher normal forms are useless or completely esoteric, but because once you have really done justice to First Normal Form and BCNF, you quite likely have it right. All of your keys are defined, and all of the nonkey columns properly reference them, but not completely. Fourth and Fifth Normal Form now focus on the relationship between key

www.it-ebooks.info

columns. If all of the natural composite key for your tables have no more than two independent key columns, you are guaranteed to be in Fourth and Fifth Normal Form if you are in BCNF as well.

Note that according to the Fourth Normal Form article in Wikipedia, there was a paper done back in 1992 by Margaret S. Wu that claimed that more than 20 percent of all databases had issues with Fourth Normal Form. And back in 1992, people actually spent time doing design, unlike today when we erect databases like a reverse implosion. However, the normal forms we will discuss in this section are truly interesting in many designs, because they center on the relationships between key columns, and both are very business-rule driven. The same table with the same columns can be a horrible mess to work with in one case, and in the next, it can be a work of art. The only way to know is to spend the time looking at the relationships.

In the next two sections, I will give an overview of Fourth and Fifth Normal Forms and what they mean to your designs. Most likely you will see some similarity to situations you have dealt with in databases before.

Fourth Normal Form: Independent Multivalued Dependencies

Fourth Normal Form deals with what are called multivalued dependencies. When we discussed dependencies in the previous sections, we discussed the case where `f(x) = y`, where both x and y were scalar values. For a multivalued dependency, the y value can be an array of values. So `f(parent) = (child1,child2, . . . , childN)` is an acceptable multivalued dependency. For a table to be in Fourth Normal Form, it needs to be in BCNF first, and then, there must not be more than one independent multivalued dependency (MVD) between the key columns.

Recall a previous example table we used:

```
Driver     VehicleStyle     DesiredModelLevel
======     ============     -----------------
Louis      CUV              Premium
Louis      Sedan            Standard
Ted        Coupe            Performance
```

Think about the key columns. The relationship between `Driver` and `VehicleStyle` represents a multivalued dependency for the `Driver` and the `VehicleStyle` entities. A driver such as Louis will drive either CUV or Sedan style vehicles, and Louis is the only driver to drive the CUV style. As we add more data, each vehicle style will have many drivers that choose the type as a preference. A table such as this one for DriverVehicleStyle is used frequently to resolve a many-to-many relationship between two tables, in this case, the `Driver` and `VehicleStyle` tables.

The modeling problem comes when you need to model a relationship between three entities, modeled as three columns in a key from three separate table types. As an example, consider the following table representing the assignment of a trainer to a class that is assigned to use a certain book:

```
Trainer    Class            Book
=======    ==============   ============================
Louis      Normalization    DB Design & Implementation
Chuck      Normalization    DB Design & Implementation
Fred       Implementation   DB Design & Implementation
Fred       Golf             Topics for the Non-Technical
```


www.it-ebooks.info

To decide if this table is acceptable, we will look at the relationship of each column to each of the others to determine how they are related. If any two columns are not directly related to one another, there will be an issue with Fourth Normal Form with the table design. Here are the possible combinations and their relationships:

- `Class` and `Trainer` are related, and a class may have multiple trainers.
- `Book` and `Class` are related, and a book may be used for multiple classes.
- `Trainer` and `Book` are not directly related, because the rule stated that the class uses a specific book.

Hence, what we really have here are two independent types of information being represented in a single table. To deal with this, you will split the table on the column that is common to the two dependent relationships. Now, take this one table and make two tables that are equivalent:

```
Class              Trainer
==============     =======
Normalization      Louis
Normalization      Chuck
Implementation     Fred
Golf               Fred

Class              Book
==============     ============================
Normalization      DB Design & Implementation
Implementation     DB Design & Implementation
Golf               Topics for the Non=Technical
```

Joining these two tables together on `Class`, you will find that you get the exact same table as before. However, if you change the book for the `Normalization` class, it will be changed immediately for both of the classes that are being taught by the different teachers. Note that initially it seems like we have more data because we have more rows and more tables. However, notice the redundancy in the following data from the original design:

```
Louis Normalization DB Design & Implementation
Chuck Normalization DB Design & Implementation
```

The redundancy comes from stating twice that the book *DB Design & Implementation* is used for the `Normalization` class. The new design conveys that same information with one less row of data. Once the system grows to the size of a full-blown system that has 50 `Normalization` classes being taught, you will have much less data, making the storage of data more efficient, possibly affording some performance benefits along with the more obvious reduction in redundant data that can get out of sync.

As an alternate situation, consider the following table of data, which might be part of the car rental system that we have used for examples before. This table defines the brand of vehicles that the driver will drive:

www.it-ebooks.info

```
Driver      VehicleStyle     VehicleBrand
======      =============    ============
Louis       Station Wagon    Ford
Louis       Sedan            Hyundai
Ted         Coupe            Chevrolet
```

- `Driver` and `VehicleStyle` are related, representing the style the driver will drive.
- `Driver` and `VehicleBrand` are related, representing the brand of vehicle the driver will drive.
- `VehicleStyle` and `VehicleBrand` are related, defining the styles of vehicles the brand offers.

This table defines the types of vehicles that the driver will take. Each of the columns has a relationship to the other, so it *is* in Fourth Normal Form. In the next section, I will use this table again to assist in identifying Fifth Normal Form issues.

Fifth Normal Form

Fifth Normal Form is a general rule that breaks out any data redundancy that has not specifically been culled out by additional rules. Like Fourth Normal Form, Fifth Normal Form also deals with the relationship between key columns. Basically, the idea is that if you can break a table with three (or more) independent keys into three (or more) individual tables and be guaranteed to get the original table by joining them together, the table is not in Fifth Normal Form

Fifth Normal Form is an esoteric rule that is only occasionally violated, but it is interesting nonetheless because it does have some basis in reality and is a good exercise in understanding how to work through intercolumn dependencies between any columns. In the previous section, I presented the following table of data:

```
Driver     VehicleStyle     VehicleBrand
======     =============    ============
Louis      Station Wagon    Ford
Louis      Sedan            Hyundai
Ted        Coupe            Chevrolet
```

At this point, Fifth Normal Form would suggest that it's best to break down any existing ternary (or greater!) relationship into binary relationships if at all possible. To determine if breaking down tables into smaller tables will be lossless (that is, not changing the data), you have to know the requirements that were used to create the table and the data. For the relationship between `Driver`, `VehicleStyle` and `VehicleBrand`, if the requirements dictate that the data is that

- Louis is willing to drive any `Station Wagon` or `Sedan` from `Ford` or `Hyundai`.
- Ted is willing to drive any `Coupe` from `Chevrolet`.

Then, we can infer from this definition of the table that the following dependencies exist:

- `Driver` determines `VehicleStyle`.
- `Driver` determines `VehicleBrand`.
- `VehicleBrand` determines `VehicleStyle`.

www.it-ebooks.info

The issue here is that if you wanted to express that Louis is now willing to drive Volvos, and that Volvo has station wagons and sedans, you would need to add least two rows:

```
Driver    VehicleStyle    VehicleBrand
======    =============   ============
Louis     Station Wagon   Ford
Louis     Sedan           Hyundai
Louis     Station Wagon   Volvo
Louis     Sedan           Volvo
Ted       Coupe           Chevrolet
```

In these two rows, you are expressing several different pieces of information. `Volvo` has `Station Wagons` and `Sedans`. Louis is willing to drive Volvos (which you have repeated multiple times). If other drivers will drive Volvos, you will have to repeat the information that Volvo has station wagons and sedans over and over.

At this point, you probably now see that ending up with tables with redundant data like in our previous example is such an unlikely mistake to make—not impossible, but not probable by any means, assuming any testing goes on with actual data in your implementation process. Once the user has to query (or worse yet, update) a million rows to express a very simple thing like the fact Volvo is now offering a sedan class automobile; changes will be made. The fix for this situation is to break the table into the following three tables, each representing the binary relationship between two of the columns of the original table:

```
Driver    VehicleStyle
======    =============
Louis     Station Wagon
Louis     Sedan
Ted       Coupe

Driver     VehicleBrand
======     ============
Louis      Ford
Louis      Hyundai
Louis      Volvo
Ted        Chevrolet

VehicleStyle      VehicleBrand
=============     ============
Station Wagon     Ford
Sedan             Hyundai
Coupe             Chevrolet
Station Wagon     Volvo
Sedan             Volvo
```

I included the additional row that says that Louis will drive Volvo vehicles and that Volvo has station wagon and sedan style vehicles. Joining these rows together will give you the table we created:

www.it-ebooks.info

```
Driver      VehicleStyle      VehicleBrand
------      -------------     ------------
Louis       Sedan             Hyundai
Louis       Station Wagon     Ford
Louis       Sedan             Volvo
Louis       Station Wagon     Volvo
Ted         Coupe             Chevrolet
```

I mentioned earlier that the meaning of the table makes a large difference. An alternate interpretation of the table could be that instead of giving the users such a weak way of choosing their desired rides (maybe Volvo has the best station wagons and Ford the best sports car), the table just presented might be interpreted as

- Louis is willing to drive Ford station wagons, Hyundai sedans, and Volvo station wagons and sedans.
- Ted is willing to drive a Chevrolet coupe.

In this case, the table is in Fifth Normal Form because instead of `VehicleStyle` and `VehicleBrand` being loosely related, they are directly related and more or less to be thought of as a single value rather than two independent ones. Now the dependency is `Driver` to `VehicleStyle` plus `VehicleBrand`. In a well-designed system, the intersection of style and brand would have formed its own table because `VehicleStyle`/`VehicleBrand` would have been recognized as an independent object with a specific key often a surrogate key which represented a `VehicleBrand` that was rentable. However, in either case, the logical representation after decoding the surrogate keys would, in fact, look just like our table of `Driver`, `VehicleStyle`, and `VehicleBrand`.

As our final example, consider the following table of `Books` along with `Authors` and `Editors`:

```
Book        Author      Editor
======      ======      ========
Design      Louis       Jonathan
Design      Jeff        Leroy
Golf        Louis       Steve
Golf        Fred        Tony
```

There are two possible interpretations that would hopefully be made clear in the name of the table:

- This table is in not even in Fourth Normal Form if it represents the following:
  - The `Book` *Design* has `Authors` Louis and Jeff and `Editors` Jonathan and Leroy.
  - The `Book` *Golf* has `Authors` Louis and Fred and `Editors` Steve and Tony.
- Table is in Fifth Normal Form if it represents
  - For the `Book` *Design*, `Editor` Jonathan edits Louis' work and `Editor` Leroy edits Jeff's work.
  - For the `Book` *Golf*, `Editor` Steve edits Louis' work, and `Editor` Tony edits Fred's work.

In the first case, the author and editor are independent of each other, meaning that technically you should have a table for the `Book` to `Author` relationship and another for the `Book` to `Editor` relationship. In the second case, the author and editor are directly related. Hence, all three of the values are required to express the single thought of "for book X, editor Y edits Z's work."

www.it-ebooks.info

■ **Note** I hope the final sentence of that previous paragraph makes it clear to you what I have been trying to say, particularly "express the single thought." Every table should represent a single thing that is being modeled. This is the goal that is being pressed by each of the normal forms, particularly the Boyce Codd, Fourth, and Fifth Normal Forms. BCNF worked through nonkey dependencies to make sure the nonkey references were correct, and Fourth and Fifth Normal Forms made sure that the key identified expressed a single thought.

What can be gleaned from Fourth and Fifth Normal Forms, and indeed all the normal forms, is that when you think you can break down a table into smaller parts with different natural keys, which then have different meanings without losing the essence of the solution you are going for, then it is probably better to do so. If you can join the parts together to represent the data in the original less-broken-down form, your database will likely be better for it. Obviously, if you can't reconstruct the table from the joins, leave it as it is. In either case, be certain to test out your solution with many different permutations of data. For example, consider adding these two rows to the earlier example:

```
VehicleStyle      VehicleBrand
=============     ============
Station Wagon     Volvo
Sedan             Volvo
```

If this data does not mean what you expected, then it is wrong. For example, if as a result of adding these rows, users who just wanted Volvo sedans were getting put into station wagons, the design would not be right.

Last, the point should be reiterated that breaking down tables ought to indicate that the new tables have different meanings. If you take a table with ten nonkey columns, you could make ten tables with the same key. If all ten columns are directly related to the key of the table, then there is no need to break the table down further.

Denormalization

Denormalization is the practice of taking a properly normalized set of tables and selectively undoing some of the changes in the resulting tables made during the normalization process for performance. Bear in mind that I said "properly normalized." I'm not talking about skipping normalization and just saying the database is denormalized. Denormalization is a process that requires you to actually normalize first, and then selectively pick out data issues that you are willing to code protection for rather than using the natural ability of normalized structures to prevent data anomalies. Too often, the term "denormalization" is used as a synonym for "work of an ignorant or, worse, lazy designer."

Back in the good old days, there was a saying: "Normalize 'til it hurts; denormalize 'til it works". In the early days, hardware was a lot less powerful, and some of the dreams of using the relational engine to encapsulate away performance issues were pretty hard to achieve. In the current hardware and software reality, there only a few reasons to denormalize when normalization has been done based on requirements and user need.

Denormalization should be used primarily to improve performance in cases where normalized structures are causing overhead to the query processor and, in turn, other processes in SQL Server or to tone down some complexity to make things easier to implement. This, of course, introduces risks of introducing data anomalies or even making the data less appropriate for the relational engine. Any additional code written to deal with these

www.it-ebooks.info

anomalies needs to be duplicated in every application that uses the database, thereby increasing the likelihood of human error. The judgment call that needs to be made in this situation is whether a slightly slower (but 100 percent accurate) application is preferable to a faster application of lower accuracy.

Denormalization should *not* be used as a crutch to make implementing the user interfaces easier. For example, say the user interface in Figure 4.14 was fashioned for a book inventory system.

Existing graphic

Does Figure 5-8 represent a bad user interface? Not in and of itself. If the design calls for the data you see in the figure to be entered, and the client wants the design, fine. However, this requirement to see certain data on the screen together is clearly a UI design issue, not a question of database structure. Don't let user interface dictate the database structure any more than the database structures should dictate the UI. When the user figures out the problem of expecting a single author for every book, you won't have to change your design.

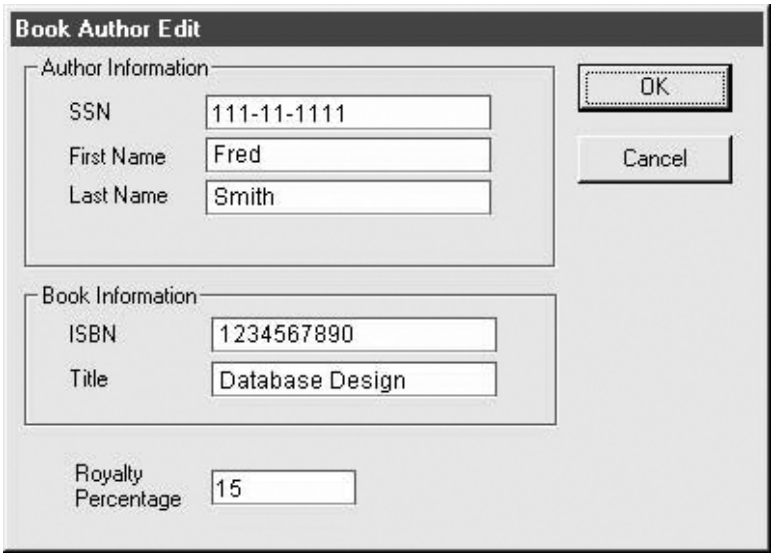


Figure 5-8. *A possible graphical front-end to our example*

■ **Note** It might also be that Figure 5-8 represents the basic UI, and a button is added to the form to implement the multiple cardinality situation of more than one author in the "expert" mode, since 90 percent of all books for your client have one author.

UI design and database design are seperate (yet interrelated) processes. The power of the UI comes with focusing on making the top 80 percent of use cases easier, and some processes can be left to be difficult if they are done rarely. The database can only have one way of looking at the problem, and it has to be as complicated as the most complicated case, even if that case happens just .1 percent of the time. If it is legal to have multiple authors, the database has to support that case, and the queries and reports that make use of the data must support that case as well.

It's my contention that during the modeling and implementation process, we should rarely step back from our normalized structures to performance-tune our applications proactively, that is to say, before a performance issue is actually felt/discovered.

www.it-ebooks.info

Because this book is centered on OLTP database structures, the most important part of our design effort is to make certain that the tables we create are well formed for the relational engine and can be equated to the requirements set forth by the entities and attributes of the logical model. Once you start the process of physical storage modeling/integration (which should be analogous to performance tuning, using indexes, partitions, etc.), there might well be valid reasons to denormalize the structures, either to improve performance or to reduce implementation complexity, but neither of these pertain to the *logical* model that represents the world that our customers live in. You will always have fewer problems if we implement physically what is true logically. For almost all cases, I always advocate waiting until you find a compelling reason to do denormalize (such as if some part of our system is failing), before we denormalize.

There is, however, one major caveat to the "normalization at all costs" model. Whenever the read/write ratio of data approaches infinity, meaning whenever data is written just once and read very, very often, it can be advantageous to store some calculated values for easier usage. For example, consider the following scenarios:

> *Balances or inventory as of a certain date:* Take a look at your bank statement. At the end of every banking day, it summarizes your activity for the day and uses that value as the basis of your bank balance. The bank never goes back and makes changes to the history but instead debits or credits the account after the balance had been fixed.
>
> *Calendar table, table of integers, or prime numbers:* Certain values are fixed by definition. For example, take a table with dates in it. Storing the name of the day of the week rather than calculating it every time can be advantageous, and given a day like November 25, 2011, you can always be sure it is a Friday.

When the writes are guaranteed to be zero, denormalization can be an easy choice, but you still have to make sure that data is in sync and cannot be made out of sync. Even minimal numbers of writes can make your implementation way too complex because again, you cannot just code for the 99.9 percent case when building a noninteractive part of the system. If someone updates a value, its copies will have to be dealt with, and usually it is far easier, and not that much slower, to use a query to get the answer than it is to maintain lots of denormalized data when it is rarely used.

One suggestion that I make to people who use denormalization as a tool for tuning an application is to always include queries to verify the data. Take the following table of data we used in an earlier section:

```
InvoiceNumber   InvoiceDate     InvoiceAmount
=============   -----------     -------------
000000000323    2011-12-23      100

InvoiceNumber    ItemNumber     Product       Quantity      ProductPrice     OrderItemId
=============    ==========     =======       ========      ============     ===========
000000000323     1              KL7R2         10            8.00             1232322
000000000323     2              RTCL3         10            2.00             1232323
```

If both the `InvoiceAmount` (the denormalized version of the summary of line item prices) are to be kept in the table, you can run a query such as the following on a regular basis during off hours to make sure that something hasn't gone wrong:

```
SELECT InvoiceNumber
FROM dbo.Invoice
GROUP BY InvoiceNumber, InvoiceAmount
```


www.it-ebooks.info

```
HAVING SUM(Quantity * ProductPrice) <> InvoiceAmount
```

Alternatively, you can feed output from such a query into the `WHERE` clause of an `UPDATE` statement to fix the data if it isn't super important that the data is maintained perfectly on a regular basis.

Best Practices

The following are a few guiding principles that I use when normalizing a database. If you understand the fundamentals of why to normalize, these five points pretty much cover the entire process:

- Follow the rules of normalization as closely as possible: This chapter's "Summary" section summarizes these rules. These rules are optimized for use with relational database management systems, such as SQL Server. Keep in mind that SQL Server now has, and will continue to add, tools that will not necessarily be of use for normalized structures, because the goal of SQL Server is to be all things to all people. The principles of normalization are 30-plus years old and are still valid today for properly utilizing the core relational engine.
- All columns must describe the essence of what's being modeled in the table: Be certain to know what that essence or exact purpose of the table is. For example, when modeling a person, only things that describe or identify a person should be included. Anything that is not directly reflecting the essence of what the table represents is trouble waiting to happen.
- At least one key must uniquely identify what the table is modeling: Uniqueness alone isn't a sufficient criterion for being a table's only key. It isn't wrong to have a uniqueness-only key, but it shouldn't be the only key.
- Choice of primary key isn't necessarily important at this point: Keep in mind that the primary key is changeable at any time with any candidate key. I have taken a stance that only a surrogate or placeholder key is sufficient for logical modeling, because basically it represents any of the other keys (hence the name "surrogate"). This isn't a required practice; it's just a convenience that must not supplant choice of a proper key.
- Normalize as far as possible before implementation: There's little to lose by designing complex structures in the logical phase of the project; it's trivial to make changes at this stage of the process. The well-normalized structures, even if not implemented as such, will provide solid documentation on the actual "story" of the data.

Summary

In this chapter, I've presented the criteria for normalizing our databases so they'll work properly with relational database management systems. At this stage, it's pertinent to summarize quickly the nature of the main normal forms we've outlined in this and the preceding chapter; see Table 5-1.

Is it always necessary to go through the steps one at a time in a linear fashion? Not exactly. Once you have designed databases quite a few times, you'll usually realize when your model is not quite right, and you'll work through the list of four things that correspond to the normal forms we have covered in this chapter.

- *Columns*: One column, one value.
- *Table/row uniqueness*: Tables have independent meaning; rows are distinct from one another.

www.it-ebooks.info

Table 5-1. *Normal Form Recap*

Form	Rules
Definition of a table	All columns must be atomic—only one value per column. All rows of a table must contain the same number of values.
First Normal Form	Every row should contain the same number of values, or in other words, no arrays, subtables, or repeating groups.
BCNF	All columns are fully dependent on a key; all columns must be a fact about a key and nothing but a key. A table is in BCNF if every determinant is a key.
Fourth Normal Form	The table must be in BCNF. There must not be more than one independent multivalued dependency represented by the table.
Fifth Normal Form	The entity must be in Fourth Normal Form. All relationships are broken down to binary relationships when the decomposition is lossless.

- *Proper relationships between columns*: Columns either are a key or describe something about the row identified by the key.
- *Scrutinize dependencies*: Make sure relationships between three values or tables are correct. Reduce all relationships to binary relationships if possible.

There is also one truth that I feel the need to slip into this book right now. You are not done. You are just starting the process of design with the blueprints for the implementation. The blueprints can and almost certainly will change because of any number of things. You may miss something the first time around, or you may discover a technique for modeling something that you didn't know before (hopefully from reading this book!), but don't get too happy yet. It is time to do some real work and build what you have designed (well, after an extra example and a section that kind of recaps the first chapters of the book, but then we get rolling, I promise).

Still not convinced? In the following list, consider the following list of pleasant side effects of normalization:

- *Eliminating duplicated data*: Any piece of data that occurs more than once in the database is an error waiting to happen. No doubt you've been beaten by this once or twice in your life: your name is stored in multiple places, then one version gets modified and the other doesn't, and suddenly, you have more than one name where before there was just one.
- *Avoiding unnecessary coding*: Extra programming in triggers, in stored procedures, or even in the business logic tier can be required to handle poorly structured data, and this, in turn, can impair performance significantly. Extra coding also increases the chance of introducing new bugs by causing a labyrinth of code to be needed to maintain redundant data.
- *Keeping tables thin*: When I refer to a "thin" table, the idea is that a relatively small number of columns are in the table. Thinner tables mean more data fits on a given page, therefore allowing the database server to retrieve more rows for a table in a single read than would otherwise be possible. This all means that there will be more tables in the system when you're finished normalizing.
- *Maximizing clustered indexes*: Clustered indexes order a table natively in SQL Server. Clustered indexes are special indexes in which the physical storage of the data matches the order of the indexed data, which allows for better performance of queries using that index. Each table can have only a single clustered index. The concept of clustered indexes applies to normalization in that you'll have more tables when you normalize. The increased numbers of clustered indexes increase the likelihood that joins between tables will be efficient.

www.it-ebooks.info

The Story of the Book So Far

This is the "middle" of the process of designing a database, so I want to take a page here and recap the process we have covered:

- You've spent time gathering information, doing your best to be thorough without going Apocalypse Now on your client. You know what the client wants, and the client knows that you know what they want.
- Next, you looked for entities, attributes, business rules, and so on in this information and drew a picture, creating a model that gives an overview of the structures in a graphical manner. (The phrase "creating a model" always makes me imagine a Frankenstein Cosmetics–sponsored beauty pageant.)
- Finally, these entities were broken down and turned into relational tables such that every table relayed a single meaning. One noun equals one table, pretty much. I'll bet if it's your first time normalizing, but not your first time working with SQL, you don't exactly love the process of normalization right now.
    - I don't blame you; it's a startling change of mind that takes time to get used to. I know the first time I had to create ten tables instead of one I didn't like it (all those little tables look troublesome the first few times!). Do it a few times, implement a few systems with normalized databases, and it will not only make sense, but you will feel unclean when you have to work with tables that aren't normalized.

If you're reading this book in one sitting (and I *hope* you aren't doing it in the bookstore without buying it), be aware that we're about to switch gears, and I don't want you to pull a muscle in your brain. We're turning away from the theory, and we're going to start doing some designs, beginning with logical designs and building SQL Server 2012 objects in reality. In all likelihood, it is probably what you thought you were getting when you first chunked down your hard-earned money for this book (or, hopefully your employer's money for a full-priced edition).

If you haven't done so, go ahead and get access to a SQL Server, such as the free SQL Server Express from Microsoft. Or download a trial copy from `http://www.microsoft.com/sql/`. Everything done in this book will work on all versions of SQL Server other than the Compact Edition.

Optionally, to do some examples, you will also need the `AdventureWorks2012` database installed for some of the examples, which you can get the latest version from `http://msftdbprodsamples.codeplex.com/`. At the time of this writing, there is a case-sensitive and case-insensitive version. I assume that you are using the case-insensitive version. I do try my best to maintain proper casing of object names.

www.it-ebooks.info

CHAPTER 6

Physical Model Implementation Case Study

Even in literature and art, no man who bothers about originality will ever be original: whereas if you simply try to tell the truth (without caring twopence how often it has been told before) you will, nine times out of ten, become original without ever having noticed it.

—C.S. Lewis

When the normalization task is complete, you have the basic structures ready for implementation, but tasks still need to be performed in the process for completing the transformation from the logical to the physical, relational model. Throughout the normalization process, you have produced legal, normalized tables that can be implemented and, using the information that should be produced during the logical modeling phase, are now ready for the finishing touches that will to turn your theoretical model into something that users (or at least developers!) can start using. At a minimum, between normalization and actual implementation, take plenty of time to review the model to make sure you are completely happy with it.

In this chapter, I'll take the normalized model and convert it into the final blueprint for the database implementation. Even starting from the same logical model, different people tasked with implementing the relational database will take a subtly (or even dramatically) different approach to the process. The final physical design will always be, to some extent, a reflection of the person/organization who designed it, although usually each of the reasonable solutions "should" resemble one another at its core.

The normalized model you have created is pretty much database agnostic and unaffected by whether the final implementation would be on Microsoft SQL Server, Microsoft Access, Oracle, Sybase, or any relational database management system. (You should expect a lot of changes if you end up implementing with a nonrelational engine, naturally.) However, during this stage, in terms of the naming conventions that are defined, the datatypes chosen, and so on, the design is geared specifically for implementation on SQL Server 2012. Each of the relational engines has its own intricacies and quirks, so it is helpful to understand how to implement on the system you are tasked with. In this book, we will stick with SQL Server.

We will go through the following steps:

- *Choosing names*: We'll look at naming concerns for tables and columns. The biggest thing here is making sure to have a standard and to follow it.

www.it-ebooks.info

- *Choosing key implementation*: Throughout the earlier bits of the book, we've made several types of key choices. In this section, we will go ahead and finalize the implementation keys for the model.
- *Determining domain implementation*: We'll cover the basics of choosing datatypes, nullability, and simple computed columns. Another decision will be choosing between using a domain table or a column with a constraint for types of values where you want to limit column values to a given set.
- *Setting up schemas*: This section provides some basic guidance in creating and naming your schemas. Beginning in SQL Server 2005, you could set up groups of tables as schemas that provide groupings of tables for usage, as well as security.
- *Adding implementation columns*: We'll consider columns that are common to almost every database that people implement that are not part of the logical design
- *Using Data Definition Language (DDL) to create the database*: In this section, we will go through the common DDL that is needed to build most every database you will encounter
- *Baseline testing your creation*: Because it's is a great practice to load some data and test your complex constraints, this section offers guidance on how you should approach and implementing testing.

■ **Note** For this and subsequent chapters, I'll assume that you have SQL Server 2012 installed on your machine. For the purposes of this book, I recommend you use the Developer Edition, which is available for a small cost from www.microsoft.com/sql/howtobuy/default.aspx. The Developer Edition gives you all of the functionality of the Enterprise Edition of SQL Server for developing software. It also includes the fully functional Management Studio for developing queries and managing your databases. (The Enterprise Evaluation Edition will also work just fine if you don't have any money to spend. Bear in mind that licensing changes are not uncommon, so your mileage may vary. In any case, there should be a version of SQL Server available to you to work through the examples.)

Another possibility is SQL Server Express Edition, which is free but doesn't come with the full complement of features of the Developer Edition. For the most part, the feature list is complete enough to use with this book. I won't make required use of any of the extended features, but if you're learning SQL Server, you'll probably want to have the full feature set to play around with. You can acquire the Express Edition in the download section at www.microsoft.com/sql/.

Finally, I'll work on a complete (if really small) database example in this chapter, rather than continue with any of the examples from previous chapters. The example database is tailored to keeping the chapter simple and to avoiding difficult design decisions, which we will cover in the next few chapters.

The main example in this chapter is based on a simple messaging database that a hypothetical company is building for its upcoming conference. Any similarities to other systems are purely coincidental, and the model is specifically created not to be overly functional but to be very, very small. The following are the simple requirements for the database:

- Messages can be 200 characters of Unicode text. Messages can be sent privately to one user, to everyone, or both. The user cannot send a message with the exact same text more than once per hour (to cut down on mistakes where users click send too often).

www.it-ebooks.info

- Users will be identified by a handle that must be 5–20 characters and that uses their conference attendee numbers and the key value on their badges to access the system. To keep up with your own group of people, apart from other users, users can connect themselves to other users. Connections are one-way, allowing users to see all of the speakers' information without the reverse being true.

Figure 6-1 shows the logical database design for this application, on which I'll base the physical design.

The following is a brief documentation of the tables and columns in the model. I won't be too specific with things like datatypes in this list. To keep things simple, I will expound on the needs as we get to each need individually.

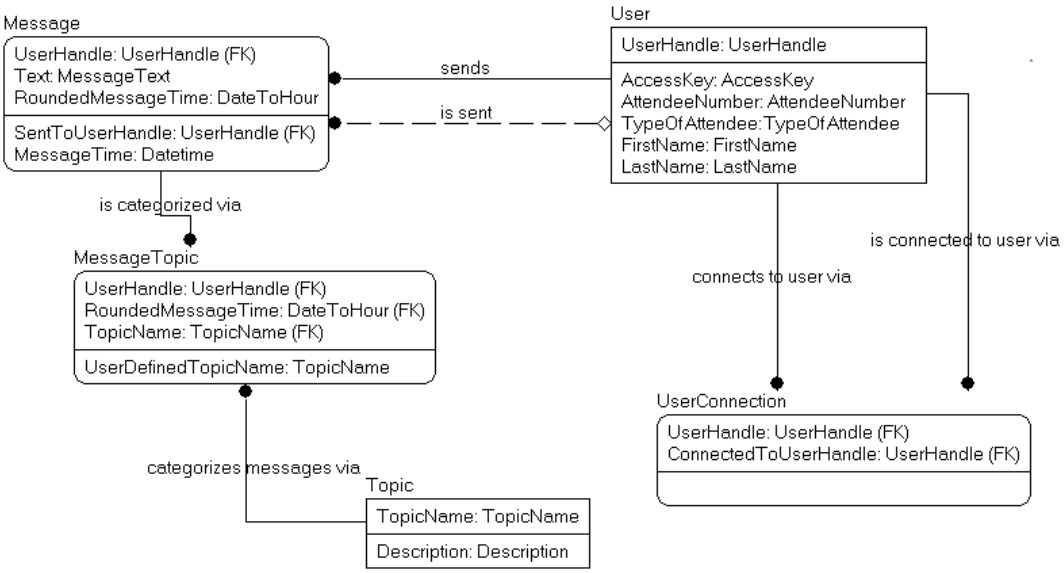


Figure 6-1. *Simple logical model of conferencing message database*

- `User`: Represents a user of the messaging system, preloaded from another system with attendee information.
  - `UserHandle`: The name the user wants to be known as. Initially pre-loaded with a value based on the persons first and last name, plus a integer value, changeable by the user.
  - `AccessKey`: A password-like value given to the users on their badges to gain access.
  - `AttendeeNumber`: The number that the attendees are given to identify themselves, printed on front of their badges.
  - `TypeOfAttendee`: Used to give the user special privileges, such as access to speaker materials, vendor areas, and so on.
  - `FirstName`, `LastName`: Name of the user printed on badge for people to see.

www.it-ebooks.info

- `UserConnection`: Represents the connection of one user to another in order to filter results to a given set of users.
  - `UserHandle`: Handle of the user who is going to connect to another user.
  - `ConnectedToUser`: Handle of the user who is being connected to.
- `Message`: Represents a single message in the system.
  - `UserHandle`: Handle of the user sending the message.
  - `Text`: The text of the message being sent.
  - `RoundedMessageTime`: The time of the message, rounded to the hour.
  - `SentToUserHandle`: The handle of the user that is being sent a message.
  - `MessageTime`: The time the message is sent, at a grain of one second.
- `MessageTopic`: Relates a message to a topic.
  - `UserHandle`: User handle from the user who sent the message.
  - `RoundedMessgeTime`: The time of the message, rounded to the hour.
  - `TopicName`: The name of the topic being sent.
  - `UserDefinedTopicName`: Allows the users to choose the `UserDefined` topic styles and set their own topics.
- `Topic`: Predefined topics for messages.
  - `TopicName`: The name of the topic.
  - `Description`: Description of the purpose and utilization of the topics.

Choosing Names

The target database for our model is SQL Server, so our table and column naming conventions must adhere to the rules imposed by this database and generally be consistent and logical. In this section, I'll briefly cover some of the different concerns when naming tables and columns. All of the system constraints on names have been the same for the past few versions of SQL Server, including 2000, 2005, and 2008.

Names of columns, tables, procedures, and so on are referred to technically as *identifiers*. Identifiers in SQL Server are stored in a system datatype of `sysname`. The system defined type named `sysname` is defined as a 128-character (or less, of course) string using double-byte Unicode characters. SQL Server's rules for identifier consist of two distinct naming methods:

- *Regular identifiers*: This is the preferred method, with the following rules:
  - The first character must be a letter as defined by Unicode Standard 3.2 (generally speaking, Roman letters A to Z, uppercase and lowercase, although this also includes other letters from other languages) or the underscore character (_). You can find the Unicode Standard at `www.unicode.org`.
  - Subsequent characters can be Unicode letters, numbers, the "at" sign (@), or the dollar sign ($).

www.it-ebooks.info

    - The name must not be a SQL Server reserved word. You can find a large list of reserved words in SQL Server 2012 Books Online, in the "Reserved Keywords" section. Some of the keywords won't cause an error, but it's better to avoid all keywords if possible. Some of these are tough, like user, transaction, and table, as they do often come up in the real world. (Note that our original model includes the name `User`, which we will have to correct.)
    - The name cannot contain spaces.
- *Delimited identifiers*: These should have either square brackets ([ ]) or double quotes ("), which are allowed only when the `SET QUOTED_IDENTIFIER` option is set to on, around the name. By placing delimiters around an object's name, you can use any string as the name. For example, `[Table Name]`, `[3232 fjfa*&(&^(]`, or `[Drop Database Master]` would be legal (but really annoying, dangerous) names. Names requiring delimiters are generally a bad idea when creating new tables and should be avoided if possible, because they make coding more difficult. However, they can be necessary for interacting with data tables in other environments. Delimiters are generally to be used when scripting objects because a name like `[Drop Database Master]` can cause "problems" if you don't.

If you need to put a closing brace (]) or even a double quote character in the name, you have to include two closing braces (]]), just like when you need to include a single quote within a string. So, the name `fred]olicious` would have to be delimited as `[fred]]olicious]`. However, if you find yourself needing to include special characters of any sort in your names, take a good long moment to consider whether you really do need this. If you determine after some thinking that you do, please ask someone else for help naming your objects, or e-mail me at louis@drsql.org. This is a pretty horrible thing to do and will make working with your objects very cumbersome. Even just including space characters is a bad enough practice that you and your users will regret for years. Note too that `[name]` and `[name ]` are treated as different names (see the embedded space). I once had a DBA name a database with a trailing space by accident . . . very annoying.

■ **Note** Using policy-based management, you can create naming standard checks for whenever a new object is created. Policy-based management is a management tool rather than a design one, though it could pay to create naming standard checks to make sure you don't accidentally create objects with names you won't accept. In general, I find doing things that way too restrictive, because there are always exceptions to the rules and automated policy enforcement only works with a dictator's hand. (Think Darth Vader, development manager!)

Table Naming

While the rules for creating an object name are pretty straightforward, the more important question is, "What kind of names should be chosen?" The answer is predictable: "Whatever you feel is best, as long as others can read it." This might sound like a cop-out, but there are more naming standards than there are data architects. (On the day this paragraph was written, I actually had two independent discussions about how to name several objects and neither person wanted to follow the same standard.) The standard I generally go with is the standard that was used in the logical model, that being Pascal-cased names, little if any abbreviation, and as descriptive as possible. With space for 128 characters, there's little reason to do much abbreviating (other than extending the life of your keyboard, I would suppose).

www.it-ebooks.info

■ **Caution** Because most companies have existing systems, it's a must to know the shop standard for naming tables so that it matches existing systems and so that new developers on your project will be more likely to understand your database and get up to speed more quickly. The key thing to make sure of is that you keep your full logical names intact for documentation purposes.

As an example, let's consider the name of the `UserConnection` table we will be building later in this chapter. The following list shows several different ways to build the name of this object:

- `user_connection` *(or sometimes, by some awful mandate, an all-caps version* `USER_CONNECTION`*)*: Use underscores to separate values. Most programmers aren't big friends of underscores, because they're cumbersome to type until you get used to them. Plus, they have a COBOLesque quality that doesn't please anyone.
- `[user connection]` **or** `"user connection"`: This name is delimited by brackets or quotes. As I have already mentioned, this isn't really favored by anyone who has done any programming, because it's impossible to use this name when building variables in code, and it's very easy to make mistakes with them. Being forced to use delimiters is annoying, and many other languages use double quotes to denote strings. (In SQL, you should always uses single quotes!) On the other hand, the brackets [and] don't denote strings, although they are a Microsoft-only convention that will not port well if you need to do any kind of cross-platform programming. Bottom line: delimited names are a bad idea anywhere except perhaps in a `SELECT` clause for a quickie report.
- `UserConnection` **or** `userConnection`: Pascal or camel case (respectively), using mixed case to delimit between words. I'll use Pascal style in the examples, because it's the style I like. (Hey, it's my book. You can choose whatever style you want!)
- `usrCnnct` **or** `usCnct`: The abbreviated forms are problematic, because you must be careful always to abbreviate the same word in the same way in all your databases. You must maintain a dictionary of abbreviations, or you'll get multiple abbreviations for the same word—for example, getting "description" as "desc," "descr," "descrip," and/or "description."

Choosing names for objects is ultimately a personal choice but should never be made arbitrarily and should be based first on existing corporate standards, then existing software, and finally legibility and readability. The most important thing to try to achieve is internal consistency. Naming, ownership, and datatypes are all things that will drive you nuts when not done consistently, because they keep everyone guessing what will be used next time. Your goal as an architect is to ensure that your users can use your objects easily and with as little thinking about structure as possible. Even most pretty bad naming conventions will be better than having ten different good ones being implemented by warring architect/developer factions. And lest you think I am kidding, in many ways the Cold War was civil compared to the internal politics of database/application design.

■ **Note** There is something to be said about the quality of corporate standards as well. If you have an archaic standard, like one that was based on the mainframe team's standard back in the 19th century, you really need to consider trying to change the standards when creating new databases so you don't end up with names like `HWWG01_TAB_USR_CONCT_T` just because the shop standards say so (and yes, I do know when the 19th century was).

www.it-ebooks.info

Naming Columns

The naming rules for columns are the same as for tables as far as SQL Server is concerned. As for how to choose a name for a column—again, it's one of those tasks for the individual architect, based on the same sorts of criteria as before (shop standards, best usage, and so on). This book follows this set of guidelines:

- Other than the primary key, my feeling is that the table name should rarely be included in the column name. For example, in an entity named `Person`, it isn't necessary to have columns called `PersonName` or `PersonSocialSecurityNumber`. Most columns should not be prefixed with the table name other than with the following two exceptions:
  - A surrogate key such as `PersonId`: This reduces the need for role naming (modifying names of attributes to adjust meaning, especially used in cases where multiple migrated foreign keys exist).
  - Columns that are naturally named with the entity name in them, such as `PersonNumber`, `PurchaseOrderNumber`, or something that's common in the language of the client and used as a domain-specific term.
- The name should be as descriptive as possible. Use few abbreviations in names, with a couple of notable exceptions:
  - *Highly recognized abbreviations*: As an example, if you were writing a purchasing system and you needed a column for a purchase-order table, you could name the object `PO`, because this is widely understood. Often, users will desire this, even if some abbreviations don't seem that obvious.
  - *Pronounced abbreviations*: If a value is read naturally as the abbreviation, then it can be better to use the abbreviation. For example, I always use `id` instead of `identifier`, first because it's a common abbreviation that's known to most people and second because the surrogate key of the Widget table is naturally pronounced Widget-Eye-Dee, not Widget-Identifier.
- Usually, the name should end in a "class" word that distinguishes the main function of the column. This class word gives a general idea of the purpose of the attribute and general expectation of datatype. It should not **be** the same thing as the datatype—for example:
  - `Store`**`Id`** is the identifier for the store.
  - `User`**`Name`** is a textual string, but whether or not it is a `varchar(30)` or `nvarchar(128)` is immaterial.
  - `End`**`Date`** is the date when something ends and does not include a time part.
  - `Save`**`Time`** is the point in time when the row was saved.
  - `Pledge`**`Amount`** is an amount of money (using a `numeric(12,2)`, or `money`, or any sort of types).
  - `Distribution`**`Description`** is a textual string that is used to describe how funds are distributed.
  - `Ticker`**`Code`** is a short textual string used to identify a ticker row.
  - `OptIn`**`Flag`** is a two-value column (possibly three including `NULL`) that indicates a status, such as in this case if the person has opted in for some particular reason.

www.it-ebooks.info

■ **Note** Many possible class words could be used, and this book is not about giving you all the standards to follow at that level. Too many variances from organization to organization make that too difficult.

I should point out that I didn't mention a Hungarian-style notation in the guidelines to denote the datatype of the column for a reason. I've never been a big fan of this style, and neither are a large number of the professional architects that I know. If you aren't familiar with Hungarian notation, it means prefixing the names of columns and variables with an indicator of the datatype and possible usage. For example, you might have a variable called `vc100_columnName` to indicate a `varchar(100)` datatype. Or you might have a Boolean or bit column named `bIsCar` or `bCarFlag`.

In my opinion, such prefixes are very much overkill, because it's easy to tell the type from other metadata you can get from SQL Server. Class word usage indicators go at the end of the name and give you enough of an idea of what a column is used for without spelling out the exact implementation details. Consider what happens if you want to change the type of a column from `varchar(100)` to `varchar(200)` because the data is of a different size than was estimated, or even because the requirements have changed. If you then have to change the name of the column, the user interface must change, the ETL to the data warehouse has to change, and all scripts and procedures have to change, even if there's no other reason to change. Otherwise, the change could be trivial, possibly needing to expand the size of only a few variables (and in some languages, even this wouldn't be required). Take the name `bCarFlag`. It is typical to use a bit datatype, but it is just as acceptable to use a `char(1)` with a domain of `'Y'` and `'N'` or any implementation that makes sense for your organization.

A particularly hideous practice that is somewhat common with people who have grown up working with procedural languages (particularly interpreted languages) is to include something in the name to indicate that a column is a column, such as `colFirstName` or `columnCity`. Please don't do this (please?). It's clear by the context in which columns are used that a column is a column. It can be used only as a column. This practice, just like the other Hungarian-style notations, makes good sense in a procedural programming language where the type of object isn't always clear just from context, but this practice is never needed with SQL tables.

■ **Note** I'll use the same naming conventions for the implementation model as I did for the logical model: Pascal-cased names with a few abbreviations (mostly in the class words, like "id" for "identifier"). I'll also use a Hungarian-style notation for objects other than tables and columns, such as constraints, and for coded objects, such as procedures. This is mostly to keep the names unique and avoid clashes with the table and column names, plus it is easier to read in a list that contains multiple types of objects (the tables are the objects with no prefixes). Tables and columns are commonly used directly by users. They write queries and build reports directly using database object names and shouldn't need to change the displayed name of every column and table.

Model Name Adjustments

In our demonstration model, the first thing we will do is to rename the `User` table to `MessagingUser` because "User" is a SQL Server reserved word. While `User` is the more natural name than `MessagingUser`, it is one of the tradeoffs we have made because of the legal values of names. In rare cases, when an unsuitable name can't be created, I may use a bracketed name, but even though it took me four hours to redraw graphics and undo my original choice of `User` as a name, I don't want to give you that as a good practice. If you find you have used a reserved word in your model (and you are not writing a chapter in a book that is 70+ pages long about it), it is usually a very minor change.

In the model snippet in Figure 6-2, I have made that change.

www.it-ebooks.info

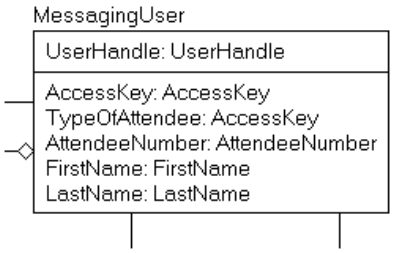


Figure 6-2. *Table User has been changed to MessagingUser*

The next change we will make will be to a few of the columns in this table. We will start off with the `TypeOfAttendee` column. The standard we discussed was to use a class word at the end of the column. In this case, `Type` will make an acceptable class, as when you see `AttendeeType`, it will be clear what it means. The implementation will be a value that will be an up to 20-character value.

The second change will be to the `AccessKey` column. `Key` itself would be acceptable as a class word, but it will give the implication that the value is a key in the database (a standard I have used in my data warehousing dimensional database designs). So adding value to the name will make the name clearer and distinctive. Figure 6-3 reflects the change in name.

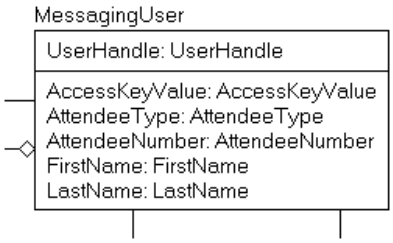


Figure 6-3. *MessagingUser table after change to AccessKey column name*

Choosing Key Implementation

The next step in the process is to choose how to implement the keys for the table. In the model at this point, it has one key identified for each table, in the primary key. In this section, we will look at the issues surrounding key choice and, in the end, will set the keys for the demonstration model. We will look at choices for implementing primary keys and then note the choices for creating alternate keys as needed.

Primary Key

Choosing the style of implementation for primary keys is an important choice. Depending on the style you go with, the look and feel of the rest of the database project will be affected. This is the case because whatever method you go with, the primary key value will be migrated to other tables as a reference to the particular row. Choosing a primary key style is also one of the most argued about topics on the forums and occasionally over dinner after a SQL Saturday event. In this book, I'll be reasonably agnostic about the whole thing, and I'll present several methods for choosing the implemented primary key throughout the book. In this chapter, I will use a very specific method, of course.

www.it-ebooks.info

Presumably, during the logical phase, you've identified the different ways to uniquely identify a row. Hence, there should be several choices for the primary key, including the following:

- Using an existing column (or set of columns)
- Deriving a new surrogate column to represent the row

Each of these choices has pros and cons. I'll look at them in the following sections.

Basing a Primary Key on Existing Columns

In many cases, a table will have an obvious, easy-to-use primary key. This is especially true when talking about independent entities. For example, take a table such as `product`. It would often have a `productNumber` defined. A person usually has some sort of identifier, either government or company issued. (For example, my company has an `employeeNumber` that I have to put on all documents, particularly when the company needs to write me a check.)

The primary keys for dependent tables can often generally take the primary key of the independent tables, add one or more attributes, and—presto!—primary key.

For example, I have a Ford SVT Focus, made by the Ford Motor Company, so to identify this particular model, I might have a row in the `Manufacturer` table for Ford Motor Company (as opposed to GM or something). Then, I'd have an `automobileMake` row with a key of `manufacturerName = 'Ford Motor Company'` and `makeName = 'Ford'` (instead of Lincoln, Mercury, Jaguar, and so on), `style = 'SVT'`, and so on, for the other values. This can get a bit messy to deal with, because the key of the `automobileModelStyle` table would be used in many places to describe which products are being shipped to which dealership. Note that this isn't about the size in terms of the performance of the key, just the number of values that make up the key. Performance will be better the smaller the key, as well, but this is true not only of the number of columns, but this also depends on the size of the values.

Note that the complexity in a real system such as this would be compounded by the realization that you have to be concerned with model year, possibly body style, different prebuilt packages, and so on. The key of the table may frequently have many parts, particularly in tables that are the child of a child of a child, and so on.

Basing a Primary Key on a New, Surrogate Value

The other common key style is to use only a single column for the primary key, regardless of the size of the other keys. In this case, you'd specify that every table will have a single primary key and implement alternate keys in your tables, as shown in Figure 6-4.

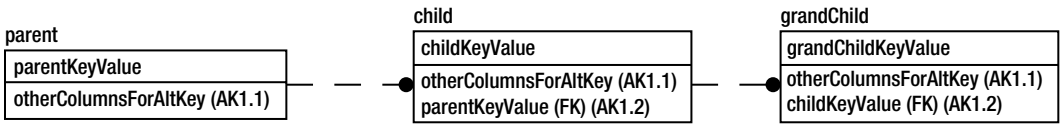


Figure 6-4. *Single-column key example*

Note that in this scenario, all of your relationships will be implemented in the database as nonidentifying type relationships, though you will implement them to all be required values (no `NULL`s). Functionally, this is the same as if the `parentKeyValue` was migrated from `parent` through `child` and down to `grandChild`, though it makes it harder to see in the model.

In the model in Figure 6-4, the most important thing you should notice is that each table not only has the primary key but also an alternate key. The term "surrogate" has a very specific meaning, even outside of

www.it-ebooks.info

computer science, and that is that it serves as a replacement. So the surrogate key for the parent object of `parentKeyValue` can be used as a substitute for the defined key, in this case `otherColumnsForAltKey`.

This method does have some useful advantages:

- *Every table has a single-column primary key*: It's much easier to develop applications that use this key, because every table will have a key that follows the same pattern. It also makes code generation easier to follow, because it is always understood how the table will look, relieving you from having to deal with all the other possible permutations of key setups.
- *The primary key index will be small*: Thus, operations that use the index to access a row in the table will be faster. Most update and delete operations will likely modify the data by accessing the data based on primary keys that will use this index.
- *Joins between tables will be easier to code*: That's because all migrated keys will be a single column. Plus, if you use a surrogate key that is named TableName + Suffix, there will be less thinking to do when setting up the join.

There are also disadvantages to this method, such as always having to join to a table to find out the meaning of the surrogate key value, plus, as in our example table in Figure 6-2, you would have to join from the `grandChild` table through the `child` table to get values from `parent`. Another issue is that some parts of the self-documenting nature of relationships are obviated, because using only single-column keys eliminates the obviousness of all identifying relationships. So in order to know that the logical relationship between `parent` and `grandchild` is identifying, you will have trace the relationship and look at the uniqueness constraints.

Assuming you have chosen to use a surrogate key, the next choice is to decide what data to use for the key. Let's look at two methods of implementing these keys, either by deriving the key from some other data or by using a meaningless surrogate value.

A popular way to define a primary key is to simply use a meaningless surrogate key like we've modeled previously, such as using a column with the `IDENTITY` property, which automatically generates a unique value. In this case, you rarely let the user have access to the value of the key but use it primarily for programming.

It's exactly what was done for most of the entities in the logical models worked on in previous chapters: simply employing the surrogate key while we didn't know what the actual value for the primary key would be. This method has one nice property:

You never have to worry about what to do when the primary key value changes.

Once the key is generated for a row, it never changes, even if all the data changes. This is an especially nice property when you need to do analysis over time. No matter what any of the other values in the table have been changed to, as long as the surrogate key value represents the same thing, you can still relate it to its usage in previous times. (This is something you have to be clear about with the DBA/programming staff as well. Sometimes, they may want to delete all data and reload it, but if the surrogate changes, your link to the unchanging nature of the surrogate key is likely broken.) Consider the case of a row that identifies a company. If the company is named Bob's Car Parts and it's located in Topeka, Kansas, but then it hits it big, moves to Detroit, and changes the company name to Car Parts Amalgamated, only one row is touched: the row where the name is located. Just change the name, and it's done. Keys may change, but not primary keys. Also, if the method of determining uniqueness changes for the object, the structure of the database needn't change beyond dropping one `UNIQUE` constraint and adding another.

Using a surrogate key value doesn't in any way prevent you from creating additional single part keys, like we did in the previous section. In fact, it pretty much demands it. For most tables, having a small code value is likely going to be a desired thing. Many clients hate long values, because they involve "too much typing." For example, say you have a value such as "Fred's Car Mart." You might want to have a code of "FREDS" for it as the shorthand

www.it-ebooks.info

value for the name. Some people are even so programmed by their experiences with ancient database systems that had arcane codes that they desire codes such as "XC10" to refer to "Fred's Car Mart."

In the demonstration model, I set all of the keys to use natural keys based on how one might do a logical model, so in a table like `MessagingUser` in Figure 6-5, it uses a key of the entire handle of the user.

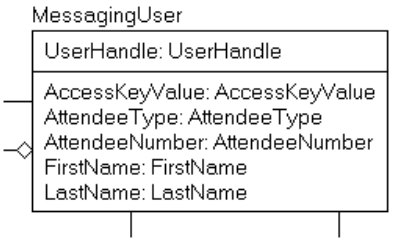


Figure 6-5. *MessagingUser table before changing model to use surrogate key*

This value is the most logical, but this name, based on the requirements, can change. Changing this to a surrogate value will make it easier to make the name change and not have to worry about existing data in the table. Making this change to the model results in the change shown in Figure 6-6, and now, the key is a value that is clearly recognizable as being associated with the `MessagingUser`, no matter what the uniqueness of the row may be. Note that I made the `UserHandle` an alternate key as I switched it from primary key.

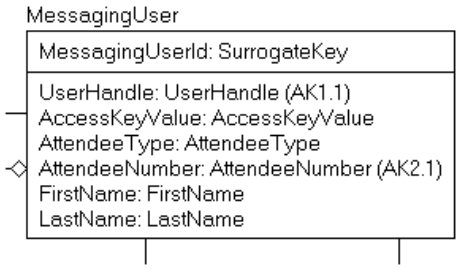


Figure 6-6. *MessagingUser table after changing model to use surrogate key*

Next up, we will take a look at the `Message` table shown in Figure 6-7. Note that the two columns that were named `UserHandle` and `SentToUserHandle` have had their role names changed to indicate the change in names from when the key of `MessagingUser` was `UserHandle`.

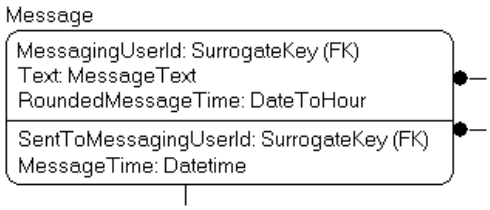


Figure 6-7. *Messaging table before changing model to use surrogate key*

www.it-ebooks.info

We will transform this table to use a surrogate key by moving all three columns to nonkey columns, placing them in a uniqueness constraint, and adding the new `MessageId` column. Notice, too, in Figure 6-8 that the table is no longer modeled with rounded corners, because the primary key no longer is modeled with any migrated keys in the primary key.

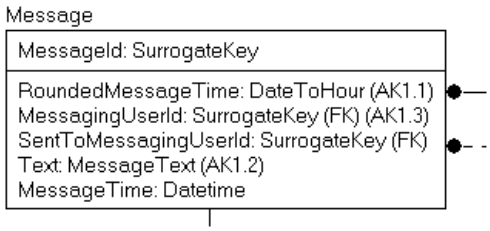


Figure 6-8. *Message table before changing model to use surrogate key*

One additional benefit of your tables having a single column surrogate key for a key is that all tables follow a common pattern. Having a common pattern for every table is useful for programming with the tables as well. Because every table has a single-column key that isn't updatable and is the same datatype, it's possible to exploit this in code, making code generation a far more straightforward process. Note once more that nothing should be lost when you use surrogate keys, because a surrogate of this style replaces an existing natural key. Many of the object relational mapping (ORM) tools that are popular (if controversial in the database community) require a single column integer key as their primary implementation pattern. I don't favor forcing the database to be designed in any manner to suit client tools, but sometimes, what is good for the database is the same as what is good for the tools, making for a relatively happy ending, at least.

By implementing tables using this pattern, I'm covered in two ways: I always have a single primary key value, but I always have a key that cannot be modified, which eases the difficulty for loading a warehouse. No matter the choice of human-accessible key, surrogate keys are the style of key that I use for all tables in databases I create, for every table. In Figure 6-9, I have completed the transformation to using surrogate keys.

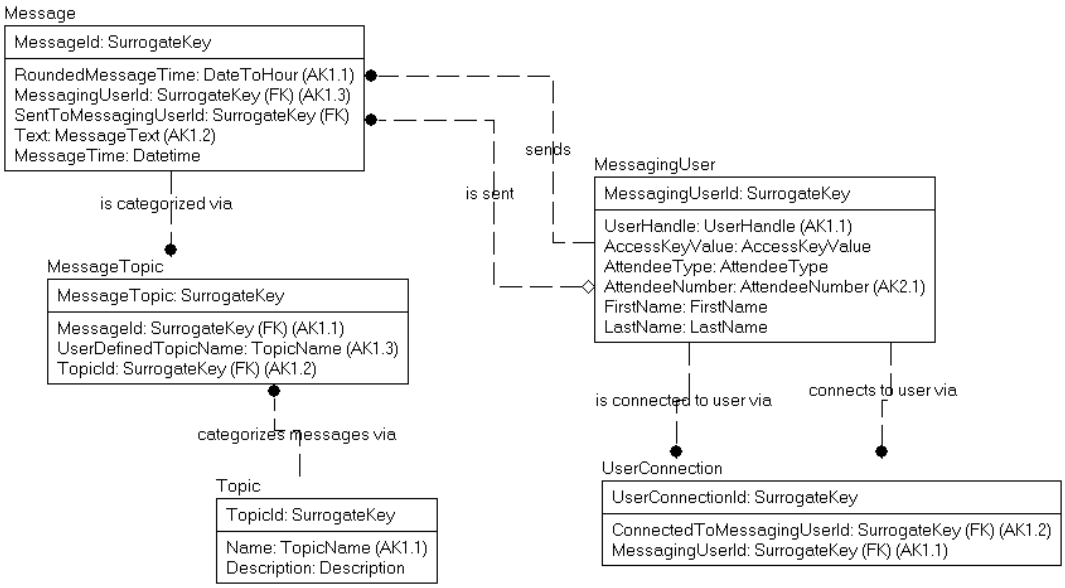


Figure 6-9. *Messaging Database Model progression after surrogate key choices*

www.it-ebooks.info

Keep in mind that I haven't specified any sort of implementation details for the surrogate key at this point, and clearly, in a real system, I would already have done this during the transformation. For this chapter example, I am using a deliberately detailed process to separate each individual step, so I will put off that discussion until the DDL section of this book, where I will present code to deal with this need along with creating the objects.

Alternate Keys

In the model so far, we have already identified alternate keys as part of the model creation (`MessagingUser.AttendeeNumber` was our only initial alternate key), but I wanted to just take a quick stop on the model and make it clear in case you have missed it. Every table should have a minimum of one natural key—that is, a key that is tied to the meaning of what the table is modeling. This step in the modeling process is exceedingly important if you have chosen to do your logical model with surrogates, and if you chose to implement with single part surrogate keys, you should at least review the keys you specified.

A primary key that's manufactured or even meaningless in the logical model shouldn't be your only defined key. One of the ultimate mistakes made by people using such keys is to ignore the fact that two rows whose only difference is a system-generated value are not different. That's because, from the user's perspective, all the data that be of value is the same. At this point, it becomes more or less impossible to tell one row from another.

For example, take Table 6-1, a snippet of a `Part` table, where `PartID` is an `IDENTITY` column and is the primary key for the table.

Table 6-1. *Sample Data to Demonstrate How Surrogate Keys Don't Make Good Logical Keys*

PartID	PartNumber	Description
1	XXXXXXXX	The X part
2	XXXXXXXX	The X part
3	YYYYYYYY	The Y part

How many individual items are represented by the rows in this table? Well, there seem to be three, but are rows with `PartID`s 1 and 2 actually the same row, duplicated? Or are they two different rows that should be unique but were keyed in incorrectly? You need to consider at every step along the way whether a human being could not pick a desired row from a table without knowledge of the surrogate key. This is why there should be a key of some sort on the table to guarantee uniqueness, in this case likely on `PartNumber`.

■ **Caution** As a rule, each of your tables should have a natural key that means something to the user and that can uniquely identify each row in your table. In the very rare event that you cannot find a natural key (perhaps, for example, in a table that provides a log of events), then it is acceptable to make up some artificial key, but usually, it is part of a larger key that helps you tell two rows apart.

In a well-designed model, you should not have anything to do at this point with keys. The architect (probably yourself) has already determined some manner of uniqueness that can be implemented. For example, in Figure 6-10, a `MessagingUser` row can be identified by either the `UserHandle` or the `AttendeeNumber`.

www.it-ebooks.info

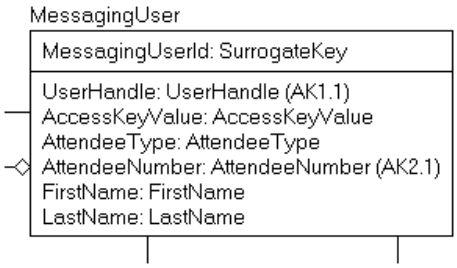


Figure 6-10. *MessagingUser table for review*

A bit more interesting is the `Message` table, shown in Figure 6-11. The key is the `RoundedMessageTime`, which is the time, rounded to the hour, the text of the message, and the `UserId`.

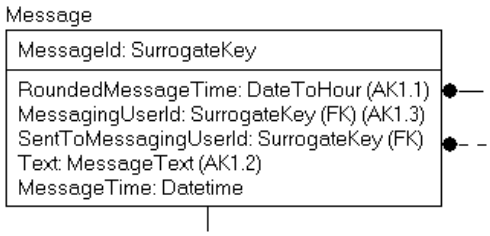


Figure 6-11. *Message table for review*

In the business rules, it was declared that the user could not post the same message more than once an hour. Constraints such as this are not terribly easy to implement in a simple manner, but breaking it down to the data you need to implement the constraint can make it easier. In our case, by putting a key on the message, user, and the time rounded to the hour, the implementation is quite easy.

Of course, by putting this key on the table, if the UI sends the same data twice, an error will be raised when a duplicate message is sent. This error will need to be dealt with at the client side, either by just ignoring the message, or translating the error message to something nicer.

The last table I will cover here is the `MessageTopic` table, shown in Figure 6-12.

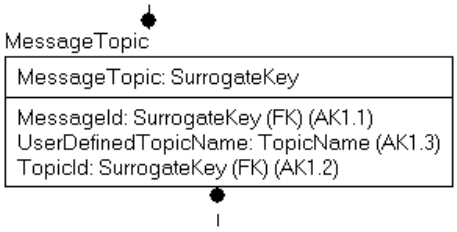


Figure 6-12. *MessageTopic table for review*

www.it-ebooks.info

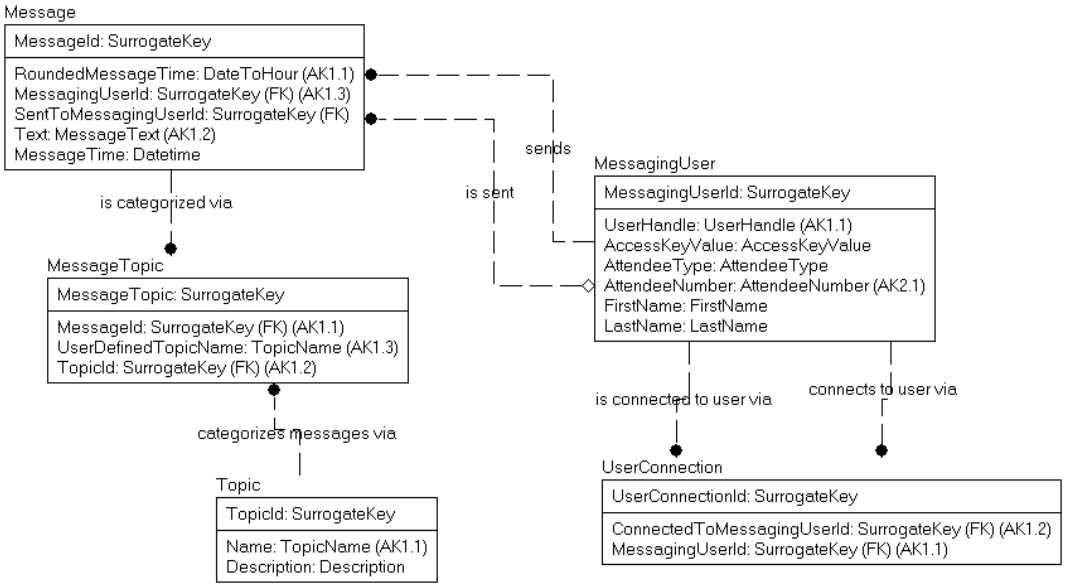


Figure 6-13. *Messaging model for review*

What is interesting about this table is the optional `UserDefinedTopicName` value. Later, when we are creating this table, we will load some seed data that indicates that the `TopicId` is `UserDefined`, which means that the `UserDefinedTopicName` column can be used. Along with this seed data, on this table will be a check constraint that indicates whether the `TopicId` value represents the user-defined topic. I will use a 0 surrogate key value. In the check constraint later, we will create a check constraint to make sure that all data fits the required criteria.

At this point, to review, we have the model at the point in Figure 6-13.

Determining Domain Implementation

In logical modeling, the concept of domains is used to specify a template for datatypes and column properties that are used over and over again. In physical modeling, domains are used to choose the datatype to use and give us a guide as to the validations we will need to implement.

For example, in the logical modeling phase, domains are defined for such columns as name and description, which occur regularly across a database/enterprise. The reason for defining domains might not have been completely obvious at the time of logical design, but it becomes clear during physical modeling. During implementation domains serve several purposes:

- *Consistency*: Define every column of type `TopicName` column in precisely the same manner; there will never be any question about how to treat the column.
- *Ease of implementation*: If the tool you use to model and implement databases supports the creation of domain and template columns, you can simply use the template to build similar columns with the same pattern, and you won't have to set the values over and over, which leads to mistakes! (Even using proper tools, I always miss some minor naming or typing issue that ends up in the final model that just irks me forever.) If you have tool support for property inheritance, when you change a property in the definition, the values change everywhere.

www.it-ebooks.info

- *Documentation*: Even if every column used a different domain and there was no reuse, the column/domain documentation would be very useful for programmers to be able to see what datatype to use for a given column. In the final section of this chapter, I will include the domain as part of the metadata I will add to the extended properties.

Domains aren't a requirement of logical or physical database design, nor does SQL Server actually make it easy for you to add them, but even if you just use them in a spreadsheet or design tool, they can enable easy and consistent design and are a great idea. Of course, consistent modeling is always a good idea regardless of whether you use a tool to do the work for you. I personally have seen a particular column type implemented in four different ways in five different columns when proper domain definitions were not available. So, tool or not, having a data dictionary that identifies columns that share a common type by definition is extremely useful.

For example, for the `TopicName` domain that's used often in the `Topic` and `MessageTopic` tables in our `ConferenceMessage` model, the domain may have been specified by the contents of Table 6-2.

Table 6-2. *Sample Domain: TopicName*

Property	Setting
Name	TopicName
Optional	No
Datatype	Unicode text, 30 characters
Value Limitations	Must not be empty string or only space characters
Default Value	n/a

I'll defer the `CHECK` constraint and `DEFAULT` bits until later in this chapter, where I discuss implementation in more depth. Several tables will have a `TopicName` column, and you'll use this template to build every one of them, which will ensure that every time you build one of these columns it will have a type of `nvarchar(30)`. Note that we will discuss data types and their exact implementation later in this chapter.

A second domain that is used very often is `SurrogateKey`, shown in Table 6-3.

Table 6-3. *Sample Domain: SurrogateKey*

Property	Setting
Name	SurrogateKey
Optional	When used for primary key, not optional, typically auto-generated. When used as a nonkey, foreign key reference, optionality determined by utilization for nonkey.
Datatype	`int`
Value Limitations	N/A
Default Value	N/A

This domain is a bit different, in that it will be implemented exactly as specified for a primary key attribute, but when it is migrated for use as a foreign key, some of the properties will be changed. First, it won't have the `IDENTITY` property set. Second, for an optional relationship, an optional relationship will allow nulls in the migrated key, but when used as the primary key, it would not allow them. Finally, let's set up one more domain definition to our sample. The `userHandle` domain, shown in Table 6-4.

www.it-ebooks.info

Table 6-4. *Sample Domain: UserHandle*

Property	Setting
Name	UserHandle
Optional	no
Datatype	Basic character set, 20 characters maximum
Value Limitations	Must be 5–20 simple alphanumeric characters and must start with a letter
Default Value	n/a

In the next four subsections, I'll discuss a couple topics concerning how to implement domains:

- *Implementing as a column or table*: You need to decide whether a value should simply be entered into a column or whether to implement a new table to manage the values.
- *Choosing the datatype*: SQL Server gives you a wide range of datatypes to work with, and I'll discuss some of the issues concerning making the right choice.
- *Choosing nullability*: In the last section, I will implement the datatype choices in the example model.
- *Choosing a collation*: The collation determines how data is sorted and compared.

Getting the datatype right is the first step in getting the implementation correct, so it can really help to spend a reasonable amount of time here making sure it is right. Too many databases end up with all datatypes the same size and nullable (except for primary keys, if they have them) and lose the integrity of having properly sized and constrained constraints.

Implement as a Column or Table?

Although many domains have only minimal limitations on values, often a domain will specify a fixed set of named values that a column might have that is less than can be fit into one of the base datatypes. For example, in the demonstration table `MessagingUser` shown in Figure 6-14, a column `AttendeeType` has a domain of `AttendeeType`.

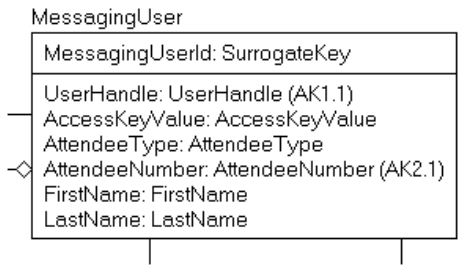


Figure 6-14. *MessageUser table for reference*

This domain might be specified as in Table 6-5.

www.it-ebooks.info

Table 6-5. *Genre Domain*

Property	Setting
Name	AttendeeType
Optional	No
Datatype	Basic character set, maximum 20 characters
Value Limitations	Regular, Volunteer, Speaker, Administrator
Default Value	Regular

The value limitation limits the values to a fixed list of values. We could choose to implement the column using a check constraint with a predicate of `AttendeeType IN ('Regular', 'Volunteer', 'Speaker', 'Administrator')` and a literal default value of `'Regular'`. There are a couple of minor annoyances with this form:

- *There is no place for table consumers to know the domain*: Unless you have a row with one of each of the values specified in the `CHECK` constraint and you do the dreaded `DISTINCT` query over the column, it isn't easy to know what the possible values are without either having foreknowledge of the system or looking in the metadata. If you're doing Conference Messaging system utilization reports by `AttendeeType`, it won't be easy to find out what `AttendeeTypes` had no activity for a time period, certainly not using a simple, straightforward SQL query that has no hard-coded values.
- *Often, a value such as this could easily have additional information associated with it*: For example, this domain might have information about actions that a given type of user could do. For example, if a `Volunteer` attendee is limited to using certain `Topics`, you would have to manage the types in a different table, with no real control on the spelling of the names. Ideally, if you define the domain value in a table, any other uses of the domain are easier to maintain.

I nearly always include tables for all domains that are essentially "lists" of items, as it is just far easier to manage, even if it requires more tables. The choice of key is a bit different than most tables, and sometimes, I use a surrogate key for the actual primary key and other times, use a natural key. The general difference is whether or not using the integer or GUID key value has value to the client's implementation. In the model, I have two examples of such types of domain implementations. In Figure 6-15, I have added a table to implement the domain for attendee types, and for this table, I will use the natural key.

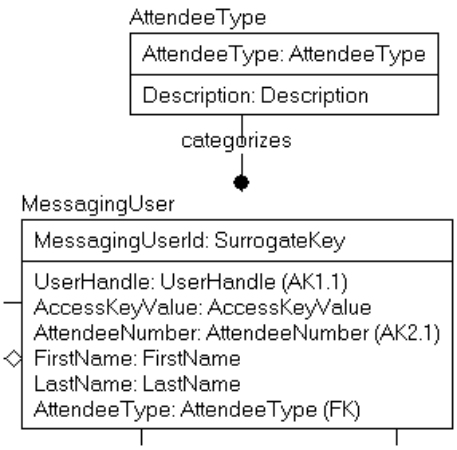


Figure 6-15. *AttendeeType domain implemented as a table*

www.it-ebooks.info

This lets an application treat the value as if it is a simple value. So if the application wants to manage the value as simple string values, I don't have to know about it from the database standpoint. I still get the value and validation that the table implementation affords me.

In the original model, we had the `Topic` table, shown in Figure 6-16, which is a domain similar to the `AttendeeType`, but that is designed to allow a user to make changes to the topic list.

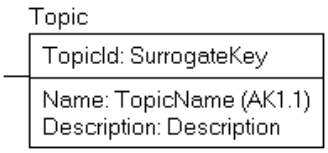


Figure 6-16. *Topic table for reference*

The `Topic` entity has the special case that it can be added to by the application managers, so it will be implemented as a numeric surrogate value. We will initialize the table with a row that represents the use defined topic that allows the user to enter their own topic in the `MessageTopic` table. Note too that we earlier discussed that the `Topic` table would loaded with a seed value for the user-defined topic.

Choosing the Datatype

Choosing proper datatypes to match the domain chosen during logical modeling is an important task. One datatype might be more efficient than another of a similar type. For example, you can store integer data in an integer datatype, a numeric datatype, a floating-point datatype or even a `varchar(10)` type, but these datatypes are certainly not alike in implementation or performance.

■ **Note** I have broken up the discussion of datatypes into two parts. First, there is this and other sections in this chapter in which I provide some basic guidance on the types of datatypes that exist for SQL Server and some light discussion on what to use. Appendix A at the end of this book is an expanded look at all of the datatypes and is dedicated to giving examples and example code snippets with all the types.

It's important to choose the best possible datatype when building the column. The following list contains the intrinsic datatypes and a brief explanation of each of them:

- *Precise numeric data*: Stores numeric data with no possible loss of precision.
  - `bit`: Stores either `1`, `0`, or `NULL`; frequently used for Boolean-like columns (1 = True, 0 = False, NULL = Unknown). Up to 8-bit columns can fit in 1 byte.
  - `tinyint`: Nonnegative values between 0 and 255 (1 byte).
  - `smallint`: Integers between –32,768 and 32,767 (2 bytes).
  - `int`: Integers between 2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31} - 1$) (4 bytes).
  - `bigint`: Integers between so it is –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (that is, -2^{63} to $2^{63} - 1$) (8 bytes).

www.it-ebooks.info

    - `decimal` *(*`numeric` *is a synonym)*: All numbers between $-10^{38} - 1$ and $10^{38} - 1$ (between 5 and 17 bytes, depending on precision).

- *Approximate numeric data*: Stores approximations of numbers, typically for scientific usage. Gives a large range of values with a high amount of precision but might lose precision of very large or very small numbers.
    - `float(N)`: Values in the range from –1.79E + 308 through 1.79E + 308 (storage varies from 4 bytes for `N` between 1 and 24, and 8 bytes for `N` between 25 and 53).
    - `real`: Values in the range from –3.40E + 38 through 3.40E + 38. `real` is a synonym for a `float(24)` datatype (4 bytes).
- *Date and time*: Stores date values, including time of day.
    - `date`: Date-only values from January 1, 0001, to December 31, 9999 (3 bytes).
    - `time`: Time-only values to 100 nanoseconds (3 to 5 bytes).
    - `datetime2(N)`: Despite the hideous name, this type will store dates from January 1, 0001, to December 31, 9999, with accuracy ranging from 1 second (0) to 100-nanosecond accuracy (7) (6 to 8 bytes).
    - `datetimeoffset`: Same as `datetime2`, but includes an offset for time zone (8 to 10 bytes).
    - `smalldatetime`: Dates from January 1, 1900, through June 6, 2079, with accuracy to 1 minute (4 bytes). (Note: it is suggested to phase out usage of this type and use the more standards oriented datetime2, though smalldatetime is not technically deprecated.)
    - `datetime`: Dates from January 1, 1753, to December 31, 9999, with accuracy to 3.33 milliseconds (8 bytes). (Note: it is suggested to phase out usage of this type and use the more standards oriented datetime2, though datetime is not technically deprecated.)
- *Binary data*: Strings of bits, for example, files or images. Storage for these datatypes is based on the size of the data stored.
    - `binary(N)`: Fixed-length binary data up to 8,000 bytes long.
    - `varbinary(N)`: Variable-length binary data up to 8,000 bytes long.
    - `varbinary(max)`: Variable-length binary data up to $(2^{31}) - 1$ bytes (2 GB) long. All the typical functionality of the varbinary columns is allowed on these types.
- *Character (or string) data*:
    - `char(N)`: Fixed-length character data up to 8,000 characters long.
    - `varchar(N)`: Variable-length character data up to 8,000 characters long.
    - `varchar(max)`: Variable-length character data up to $(2^{31}) - 1$ bytes (2 GB) long. All the typical functionality of the `varchar` columns is allowed on these types.
    - `nchar(N)`, `nvarchar(N)`, `nvarchar(max)`: Unicode equivalents of `char`, `varchar`, and `varchar(max)`.

www.it-ebooks.info

- *Other datatypes*:
  - `sql_variant`: Stores any datatype. It's generally a bad idea to use this datatype, but it is handy in cases where you don't know the datatype of a value before storing. Best practice would be to describe the type in your own metadata when using this type.
  - `rowversion` *(*`timestamp` *is a synonym)*: Used for optimistic locking to version-stamp a row. It changes on every modification. The name of this type was `timestamp` in all SQL Server versions before 2000, but in the ANSI SQL standards, the `timestamp` type is equivalent to the datetime datatype. I'll demonstrate the rowversion datatype in detail in Chapter 10, which is about concurrency.
  - `uniqueidentifier`: Stores a GUID value.
  - `XML`: Allows you to store an XML document in a column. The `XML` type gives you a rich set of functionality when dealing with structured data that cannot be easily managed using typical relational tables. You shouldn't use the `XML` type as a crutch to violate the First Normal Form by storing multiple values in a single column. I will not use `XML` in any of the designs in this book.
  - Spatial types (`geometry`, `geography`, `circularString`, `compoundCurve`, and `curvePolygon`): Used for storing spatial data, like for maps. I will not be using this type in this book.
  - `heirarchyId`: Used to store data about a hierarchy, along with providing methods for manipulating the hierarchy. We will cover more about manipulating hierarchies in Chapter 8.

Choice of datatype is a tremendously important part of the process, but if you have defined the domain well, it is not that difficult of a task. In the following sections, we will look at a few of the more important parts of the choice. A few of the considerations we will include are

- Deprecated or bad choice types
- Common datatype configurations
- Large-value datatype columns
- Complex datatypes

I didn't use too many of the different datatypes in the sample model, because my goal was to keep the model very simple and not try to be an AdventureWorks-esque model that tries to show every possible type of SQL Server in one model. In the next chapters of patterns, we will include a good amount of the datatypes in our examples because there are good pattern usages of almost all of the common types.

Deprecated or Bad Choice Types

I didn't include several datatypes in the previous list listed because they have been deprecated for quite some time, and it wouldn't be surprising if they were completely removed from the version after 2012, even though I said the same thing in the previous version of the book so be sure to stop using them as soon as possible). Their use was common in versions of SQL Server before 2005, but they've been replaced by types that are *far* easier to use:

- `image`: Replace with `varbinary(max)`
- `text` **or** `ntext`: Replace with `varchar(max)` and `nvarchar(max)`

www.it-ebooks.info

If you have ever tried to use the text datatype in SQL code, you know it is not a pleasant thing. Few of the common text operators were implemented to work with it, and in general, it just doesn't work like the other native types for storing string data. The same can be said with image and other binary types. Changing from `text` to `varchar(max)`, and so on, is definitely a no-brainer choice.

The second types that are generally advised against being used are the two money types:

- `money`: `-922,337,203,685,477.5808` through `922,337,203,685,477.5807` (8 bytes)
- `smallmoney`: Money values from `-214,748.3648` through `214,748.3647` (4 bytes)

In general, the `money` datatype sounds like a good idea, but using has some confusing consequences. In Appendix A, I spend a bit more time covering these consequences, but here are two problems:

- There are definite issues with rounding off, because intermediate results for calculations are calculated using only four decimal places.
- Money data output includes formatting, including a monetary sign (such as $ or £), but inserting `$100` and `£100` results in the same value being represented in the variable or column.

Hence, it's generally accepted that it's best to store monetary data in decimal datatypes. This also gives you the ability to assign the numeric types to sizes that are reasonable for the situation. For example, in a grocery store having the maximum monetary value of a grocery item over 200,000 dollars is probably unnecessary, even figuring for a heck of a lot of inflation. Note that in Appendix A I will include a more thorough example of the types of issues you will see.

Common Datatype Configurations

In this section, I will briefly cover concerns and issues relating to Boolean/logical values, large datatypes, and complex types and then summarize datatype concerns in order to discuss the most important thing you need to know about choosing a datatype.

Boolean/Logical Values

Booleans are another of the hotly debated choices that are made for SQL Server data. There's no Boolean type in standard SQL, since every type must support `NULL`, and a `NULL` Boolean makes life far more difficult for the people who implement SQL, so a suitable datatype needs to be chosen through which to represent Boolean values. Truthfully, though, what we really want from a Boolean is the ability to say that the property of the modeled entity "is" or "is not" for some basic setting.

There are three common choices to implement a value of this sort:

- *Using a* `bit` *datatype where a value of* `1:True` *and* `0:False`: This is, by far, the most common datatype because it works directly with programming languages such as VB .NET with no translation. The check box and option controls can directly connect to these values, even though VB uses `-1` to indicate `True`. It does, however, draw the ire of purists, because it is too much like a Boolean. Commonly named "flag" as a class word, like for a special sale indicator: `SpecialSaleFlag`. Some people who don't do the suffix thing as a rule often start the name off with `Is`, like `IsSpecialSale`. Microsoft uses the prefix in the catalog views quite often, like in `sys.databases`: `is_ansi_nulls_on`, `is_read_only`, and so on.
- *A* `char(1)` *value with a domain of* `'Y'`, `'N'`; `'T'`, `'F'`, *or other values*: This is the easiest for ad hoc users who don't want to think about what `0` or `1` means, but it's generally the most difficult from a programming standpoint. Sometimes, a `char(3)` is even better to go with `'yes'` and `'no'`. Usually named the same as the bit type, but just having a slightly more attractive looking output.

www.it-ebooks.info

- *A full, textual value that describes the need*: For example, a preferred customer indicator, instead of `PreferredCustomerFlag`, `PreferredCustomerIndicator`, with values `'Preferred Customer'` and `'Not Preferred Customer'`. Popular for reporting types of databases, for sure, it is also more flexible for when there becomes more than 2 values, since the database structure needn't change if you needed to add `'Sorta Preferred Customer'` to the domain of `PreferredCustomerIndicator`.

As an example of a Boolean column in our messaging database, I'll add a simple flag to the `MessagingUser` table that tells whether the account has been disabled, as shown in Figure 6-17. As before, we are keeping things simple, and in simple cases, a simple flag might do it. But of course, in a sophisticated system, you would probably want to have more information, like who did the disabling, and why they did it.

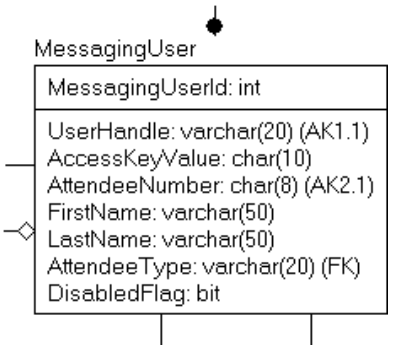


Figure 6-17. *MessagingUser table with DisabledFlag bit column*

Large-Value Datatype Columns

As of SQL Server 2005, dealing with large datatypes changed quite a bit. By using the `max` specifier on `varchar`, `nvarchar`, and `varbinary` types, you can store far more data than was possible in previous versions using a "normal" type, while still being able to deal with the data using the same functions and techniques you can on a simple `varchar(10)` column, though performance will differ slightly.

As with all datatype questions, use the `varchar(max)` types only when they're required, and you should always use the smallest types possible. The larger the datatype, the more data possible, and the more trouble the row size can be to get optimal storage retrieval times. In cases where you know you need large amounts of data or in the case where you sometimes need greater than 8,000 bytes in a column, the `max` specifier is a fantastic thing.

■ **Note** Keep on the lookout for uses that don't meet the normalization needs. Most databases have a "comments" column somewhere that morphs from comments to a semistructured mess that your DBA staff then needs to dissect using the dreaded `SUBSTRING` and `CHARINDEX` functions.

There are two special concerns when using these types:

- There's no automatic datatype conversion from the normal character types to the large-value types.
- Because of the possible large sizes of data, a special clause is added to the `UPDATE` statement.

www.it-ebooks.info

The first issue is pretty simple, but it can be a bit confusing at times. For example, concatenate `'12345' + '67890'`. You've taken two `varchar(5)` values, and the result will be contained in a value that is automatically be recast as a `varchar(10)`. But if you concatenate two `varchar(8000)` values, you don't get a `varchar(16000)` value, and you don't get a `varchar(max)` value. The values get truncated to a `varchar(8000)` value. This isn't always intuitively obvious. For example, consider the following code:

```
SELECT  LEN(CAST(replicate('a',8000) AS varchar(8000))
          + CAST(replicate('a',8000) AS varchar(8000))
        );
```

It returns a value of type `varchar(8000)`. If you cast one of the `varchar(8000)` values to `varchar(max)`, then the result will be 16,000:

```
SELECT  LEN(CAST(replicate('a',8000) AS varchar(max))
          + CAST(replicate('a',8000) AS varchar(8000))
        );
```

Second, because the size of columns stored using the `varchar(max)` datatype can be so huge, it wouldn't be favorable to always pass around these values just like you do with smaller values. Because the maximum size of a `varchar(max)` value is 2 GB, imagine having to update a value of this size in its entirety. Such an update would be pretty nasty, because the client would need to get the whole value, make its changes, and then send the value back to the server. Most client machines may only have 2 GB of physical RAM, so paging would like occur on the client machine, and the whole process would crawl and probably crash. So, you can do what are referred to as **chunked** updates. These are done using the `.WRITE` clause in the `UPDATE` statement. For example

```
UPDATE TableName
SET    varcharMaxCol.WRITE('the value', <offset>, <expression>)
WHERE  . . .
```

One important thing to note is that `varchar(max)` values will easily cause the size of rows to go greater than the 8060-byte limit with the data placed on overflow pages. Overflow pages are not terribly efficient because SQL Server has to go fetch extra pages that will not be in line with other data pages. (Overflow pages are covered more in Chapter 10 when the physical structures are covered.)

I won't go over large types in any more detail at this point. Just understand that you might have to treat the data in the `(max)` columns differently if you're going to allow large quantities of data to be stored. In our model, we've used a `varbinary(max)` column in the `Customer` table to store the image of the customer.

The main point to understand here is that having a datatype with virtually unlimited storage comes at a price. SQL Server 2008 allows you some additional freedom when dealing with `varbinary(max)` data by placing it in the file system using what is called **filestream storage**. I will discuss large object storage in Chapter 8 in more detail, including filestreams.

User Defined Type/Alias

One really excellent sounding feature that you can use to help make your code more clean is a user defined type, which is really an alias to a type. I don't want to get too much into syntax yet, but you can use the datatype alias to specify a commonly used datatype configuration that's used in multiple places using the following syntax:

```
CREATE TYPE <typeName>
  FROM <intrinsic type> --any type that can be used as a column of a
                        --table, with precision and scale or length,
                        --as required by the intrinsic type
  [NULL | NOT NULL]
```


www.it-ebooks.info

When declaring a table, if nullability isn't specified, then `NULL` or `NOT NULL` is based on the setting of `ANSI_NULL_DFLT_ON`, except when using a alias type (variables will always be nullable). In general, it is best to always specify the nullability in the table declaration.

For example, consider the `UserHandle` column. Earlier, we defined its domain as being `varchar(20)`, not optional, alphanumeric, with the data required to be between 5 and 20 characters. The datatype alias would allow us to specify:

```
CREATE TYPE UserHandle FROM varchar(20) NOT NULL;
```

Then, in the table create statement, we could specify

```
CREATE TABLE MessagingUser
…
UserHandle UserHandle,
```

By declaring that the `UserHandle` type will be `varchar(20)`, you can ensure that every time the type of `UserHandle` is used, in table declarations, and variable declarations will be `varchar(20)` and as long as you don't specify `NULL` or `NOT NULL`. It is not possible to implement the requirement that data be between 5 and 20 characters on any other constraints on the type, including the null specification.

For another example, consider an `SSN` type. It's `char(11)`, so you cannot put a 12-character value in, sure. But what if the user had entered `234433432` instead of including the dashes? The datatype would have allowed it, but it isn't what's desired. The data will still have to be checked in other methods such as `CHECK` constraints.

I am personally not a user of these types. I have never really used these kinds of types, because of the fact that you cannot do anything with these other than simply alias a type. Any changes to the type also require removal of all references to the type.

I will note, however, that I have a few architect friends who make extensive use of them to help keep data storage consistent. I have found that using domains and a data modeling tool serves me a bit better, but I do want to make sure that you have at least heard of them and know the pros and cons.

Complex Datatypes

In SQL Server 2005 and later, we can build our own datatypes using the SQL CLR. Unfortunately, they are quite cumbersome, and the implementation of these types does not lend itself to the types behaving like the intrinsic types. Utilizing CLR types will require you to install the type on the client for them to get the benefit of the type being used.

Hence, while it is possible to do, for the most part you should use them only in the cases where it makes a very compelling reason to do so. There are a few different possible scenarios where you could reasonably use user-defined types (UDTs) to extend the SQL Server type system with additional scalar types or different ranges of data of existing datatypes. Some potential uses of UDTs might be:

- Complex types that are provided by an owner of a particular format, such as a media format that could be used to interpret a `varbinary(max)` value as a movie or an audio clip. This type would have to be loaded on the client to get any value from the datatype.
- Complex types for a specialized application that has complex needs, when you're sure your application will be the only user.

Although the possibilities are virtually unlimited, I suggest that CLR UDTs be considered only for specialized circumstances that make the database design extremely more robust and easy to work with. CLR UDTs are a nice addition to the DBA's and developer's toolkit, but they should be reserved for those times when adding a new scalar datatype solves a business problem.

In SQL Server 2008, Microsoft provided several CLR user-defined types to implement hierarchies and spatial datatypes. I point this out here to note that if Microsoft is using the CLR to implement complex types (and the

www.it-ebooks.info

spatial types at the very least are pretty darn complex), the sky is the limit. I should note that the spatial and `hierarchyId` types push the limits of what should be in a type, and some of the data stored (like a polygon) is really an array of connected points.

The Most Important Consideration for Choosing Datatypes

When all is said and done, the most important consideration when choosing a datatype is to keep things simple and choose the right types for the job. SQL Server gives you a wide range of datatypes, and many of them can be declared in a wide variety of sizes. I never cease to be amazed by the number of databases around where every single column is either an integer or a `varchar(N)` (where `N` is the same for every single string column) and `varchar(max)`. One particular example I've worked with had everything, including GUID-based primary keys, all stored in `nvarchar(200)` columns! It is bad enough to store your GUIDs in a `varchar` column at all, since it is stored as a 16-byte binary value, and as a `varchar` column, it will take 36 bytes; however, store it in an `nvarchar` column, and now, it takes at least 72 bytes! What a hideous waste of space. Even worse, now all data could be up to 200 characters wide, even if you plan to give entry space for only 30 characters. Now, people using the data will feel like they need to allow for 200 characters on reports and such for the data. Time wasted, space wasted, money wasted.

As another example, say you want to store a person's name and date of birth. You could choose to store the name in a `varchar(max)` column and the date of birth in a `varchar(max)` column. In all cases, these choices would certainly store the data that the user wanted, but they wouldn't be good choices. The name should be in something such as a `varchar(30)` column and the date of birth in a date column. Notice that I used a variable size type for the name. This is because you don't know the length and not all names are the same size. Because most names aren't nearly 30 bytes, using a variable-sized type will save space in your database.

Of course, in reality, seldom would anyone make such poor choices of a datatype as putting a date value in a `varchar(max)` column. Most choices are reasonably easy. However, it's important keep in mind that the datatype is the first level of domain enforcement. Thinking back to our domain for `UserHandle`, we had the following datatype definition, and value limitations:

Table 6-6. *Sample Domain:UserHandle*

Property	Setting
Name	UserHandle
Optional	no
Datatype	Basic character set, maximum 20 characters
Value Limitations	Must be between 5-20 characters, simple alphanumeric and start with a letter
Default Value	n/a

You can enforce the first part of this at the database level by declaring the column as a `varchar(20)`. A column of type `varchar(20)` won't even allow a 21-character or longer value to be entered. It isn't possible to enforce the rule of greater than or equal to five characters using only a datatype. I'll discuss more about how to enforce simple domain requirements later in this chapter, and in Chapter 9, we will discuss patterns of integrity enforcements that are more complex.

In the earlier part of the process, we defined domains for every one of our columns (well, theoretically, in actuality some of them are simply named now, but we will make assumptions about each column in this chapter, so we can bring it in under 200 pages).

Initially, we had the model in Figure 6-18 for the `MessagingUser` table.

www.it-ebooks.info

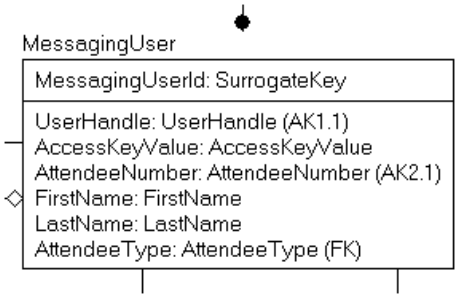


Figure 6-18. *MessagingUser table before choosing exact datatypes*

Choosing types, we will use an `int` for the surrogate key (and in the DDL section, we will set the implementation of the rest of the optionality rule set in the domain: "Not optional auto generated for keys, optionality determined by utilization for nonkey", but will replace items of `SurrogateKey` domain with `int` types. User handle was discussed earlier in this section. In Figure 6-19, I chose some other basic types for `Name`. `AccessKeyValue`, and the `AttendeeType` columns.

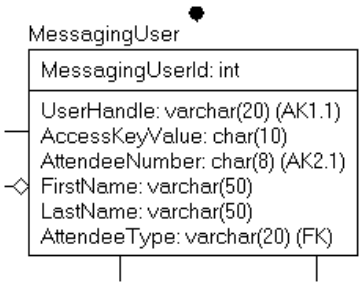


Figure 6-19. *MessagingUser after datatype choice*

Sometimes, you won't have any real domain definition, and you will use common sizes. For these, I suggest using either a standard type (if you can find them, like on the Internet) or look through data you have in your system. Until the system gets into production, changing types is fairly easy from a database standpoint, but the more code that accesses the structures the more difficult it gets to make changes.

For the `Message` table in Figure 6-20, we will choose types.

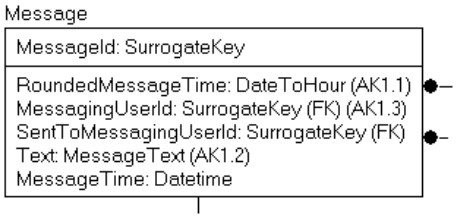


Figure 6-20. *Message table before datatype choice*

www.it-ebooks.info

The text column isn't datatype text but is the text of the message, limited to 200 characters. For the time columns, in Figure 6-21, I choose `datetime2(0)` for the `MessageTime`, since the requirements specified time down to the second. For `RoundedMessageTime`, we are rounding to the hour, and so I chose also will expect it to be `datetime2(0)`, though it will be a calculated column based on the `MessageTime` value. Hence, `MessageTime` and `RoundedMessageTime` are two views of the same data value.

So I am going to use a calculated column as shown in Figure 6-21, I will specify the type of `RoundedMessageTime` as a nonexistent datatype (so if I try to create the table it will fail). A calculated column is a special type of column that isn't directly modifiable, as it is based on the result of an expression.

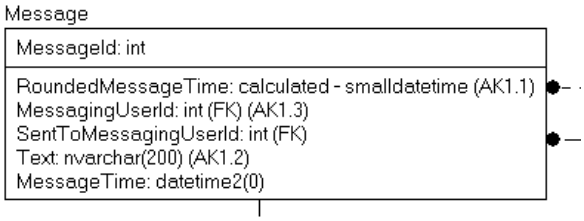


Figure 6-21. *Message table after datatype choice, with calculated column denoted*

Later in this chapter, we will specify the actual implementation, but for now, we basically just set a placeholder. Of course, in reality, I would specify the implementation immediately, but again, for this first learning process, I am doing things in this deliberate manner to keep things orderly. So, in Figure 6-22, I have the model with all of the datatypes set.

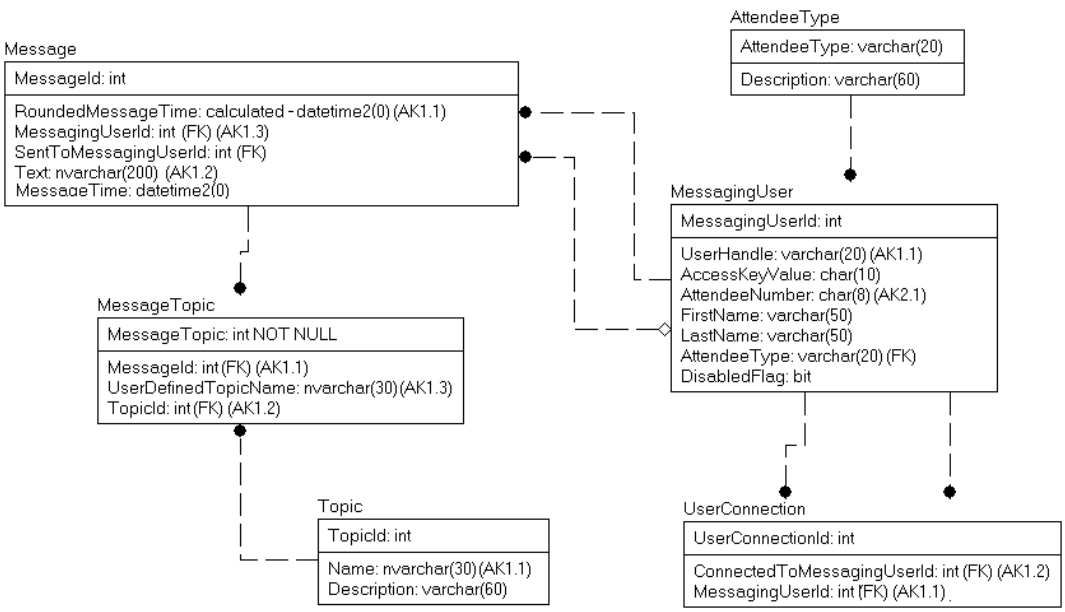


Figure 6-22. *Messaging system model after datatype choices*

www.it-ebooks.info

Choosing Nullability

The next step in the process is to set nullability of columns. In our domains, we specified if the columns were optional, so this will generally be a simple task. For the `Message` table in Figure 6-23, I have chosen the following nullabilities.

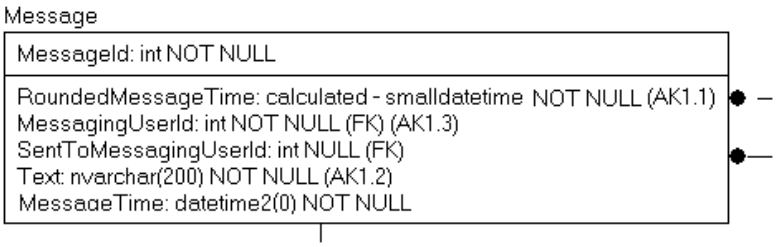


Figure 6-23. *Message table for review*

The interesting choice was for the two `MessagingUserId` columns. In Figure 6-24, you can see the full model, but note the relationships from `MessagingUser` to `Message`. The relationship for the user that sent the message (`MessagingUserId`) is `NOT NULL`, because every message is sent by a user. However, the relationship representing the user the message was sent to is nullable, since not every message needs to be sent to a user.

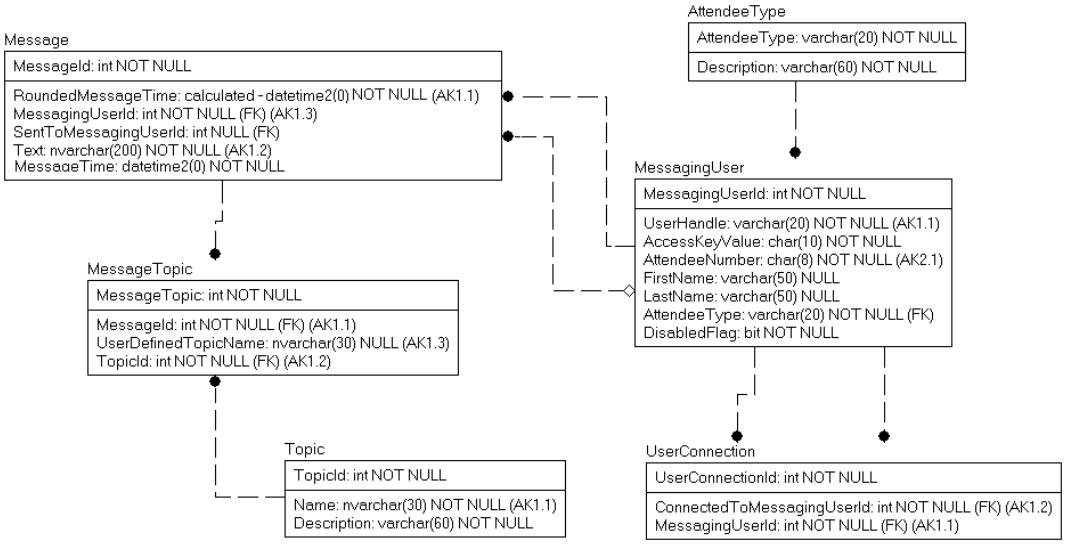


Figure 6-24. *Messaging System model, with NULLs chosen*

At this point, our model is very nearly done and very much resembles a database that could be used. Just a bit more information needed to finish out the model.

www.it-ebooks.info

Choosing a Collation

Many character sets are used by the many different cultures around the world. While you can choose a Unicode datatype if you need to store the characters for almost any character set, there still is the question of how data is sorted (case sensitive or not) and compared (accent sensitive or not). The collation sequence shows how data is sorted when needed and how data is compared. SQL Server and Windows provide a tremendous number of collation types to choose from. The collation is specified at many levels, starting with the server. The server collation determines how much of the system metadata is stored. Then the database has a collation, and finally, each column may have a different collation.

It's a somewhat uncommon need for the average database to change the collation from the default, which is usually chosen to be the most useful for most uses. This is usually a case-insensitive collation, which allows that when doing comparisons and sorts, `'A' = 'a'`. I've only used an alternative collation a few times for columns where case sensitivity was desired (one time was so that a client could force more four-character codes than a case-insensitive collation would allow!).

To see the current collation type for the server and database, you can execute the following commands:

```
SELECT serverproperty('collation');
SELECT databasepropertyex('MovieRental','collation');
```

On most systems installed in English-speaking countries, the default collation type is `SQL_Latin1_General_CP1_CI_AS`, where `Latin1_General` represents the normal Latin alphabet, `CP1` refers to code page 1252 (the SQL Server default Latin 1 ANSI character set), and the last parts represent case insensitive and accent sensitive, respectively. You can find full coverage of all collation types in the SQL Server 2012 documentation.

To list all the sort orders installed in a given SQL Server instance, you can execute the following statement:

```
SELECT *
FROM fn_helpcollations();
```

On the computer on which I do testing, this query returned more than 1,000 rows, but usually, you don't need to change from the default that the database administrator initially chooses. To set the collation sequence for a `char`, `varchar`, `text`, `nchar`, `nvarchar`, or `ntext` column when creating a column, you specify it using the `COLLATE` clause of the column definition, like so:

```
CREATE TABLE alt.OtherCollate
(
    OtherCollateId integer IDENTITY
        CONSTRAINT PKAlt_OtherCollate PRIMARY KEY ,
    Name nvarchar(30) NOT NULL,
    FrenchName nvarchar(30) COLLATE French_CI_AS_WS NULL,
    SpanishName nvarchar(30) COLLATE Modern_Spanish_CI_AS_WS NULL
);
```

Now, when you sort output by `FrenchName`, it's case insensitive, but arranges the rows according to the order of the French character set. The same applies with Spanish, regarding the `SpanishName` column. For this chapter we will stick with the default, and I would suggest taking a look at books online if you have the need to store data in multiple languages.

One quick note, you can specify the collation in a `WHERE` clause using the `COLLATE` keyword:

```
SELECT Name
FROM alt.OtherCollate
WHERE Name COLLATE Latin1_General_CS_AI
        LIKE '[A-Z]%' collate Latin1_General_CS_AI; --case sensitive and
                                                    --accent insensitive
```


www.it-ebooks.info

It is important to be careful when choosing a collation that is different from the default because at the server level it is extremely hard to change, and at the database level it is no picnic. You can change the collation of a column with an `ALTER` command, but it can't have constraints or indexes referencing it, and you may need to recompile all of your objects that reference the tables.

Setting Up Schemas

A **schema** is a namespace: a container where database objects are contained, all within the confines of a database. We will use them to group our tables in functional groups. Naming schemas is a bit different than tables or columns. Schema names should sound right, so sometimes, they make sense to be plural, and other times singular. It depends on how they are being used. I find myself using plural names most of the time because it sounds better, and because sometimes, you will have a table named the same thing as the schema if both were singular.

In our model in Figure 6-25, we will put the tables that are used to represent messages in a `Messages` schema, and the ones that represent `Attendees` and their relationships to one another in a schema we will name `Attendees`.

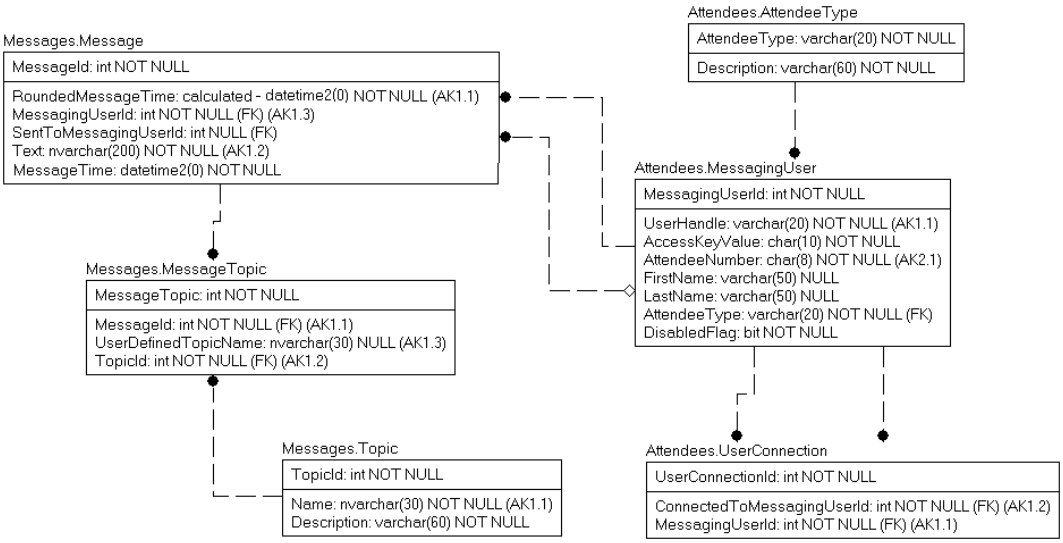


Figure 6-25. *Messages model with schemas assigned*

Note, too, that I often will set up schemas late in the process, and it might seem more correct to start there. I find that it is often easier to discover the different areas of the implementation, and that schemas aren't necessarily easy to start with, but that different areas come and go until I get to the final solution. Sometimes, it is by necessity because you have multiple tables with the same name, though this can be a sign of a bad design. In this manufactured solution, I simply did it last to make the point that it could be last.

What makes schemas so nice is that you can deal with permissions on a schema level, rather than on an object-by-object level. Schemas also give you a logical grouping of objects when you view them within a list, such as in Management Studio.

www.it-ebooks.info

I'm not going to go any further into the security aspects of using schemas at this point in the book, but I'll just mention that they're a good idea. Throughout this book, I'll always name the schema that a table is in when doing examples. Schemas will be part of any system I design in this book, simply because it's going to be best practice to do so going further. On a brief trip back to the land of reality, I would expect that beginning to use schemas in production systems will be a slow process, because it hasn't been the normal method in years past. Chapter 9 will discuss using schemas for security in more detail.

Adding Implementation Columns

Finally, I will add one more thing to the database, and these are columns to support the implementation. A very common use is to have columns that indicate when the row was created, when it was updated, by whom, and so on. In our model, I will stick to the simple case of the times mentioned and will demonstrate how to implement this in the database. A lot of implementers like to leave these values to the client, but I very much prefer using the database code because then we have one clock managing times, rather than multiples.

So in Figure 6-26, I add two `NOT NULL` columns to every table for the `RowCreateTime` and `RowLastUpdateTime`, except for the `AttendeeType` table. As we specified it to be not user manageable, I chose not to include the modified columns for that table. Of course, you might want to do this to let the user know when the row was first available.

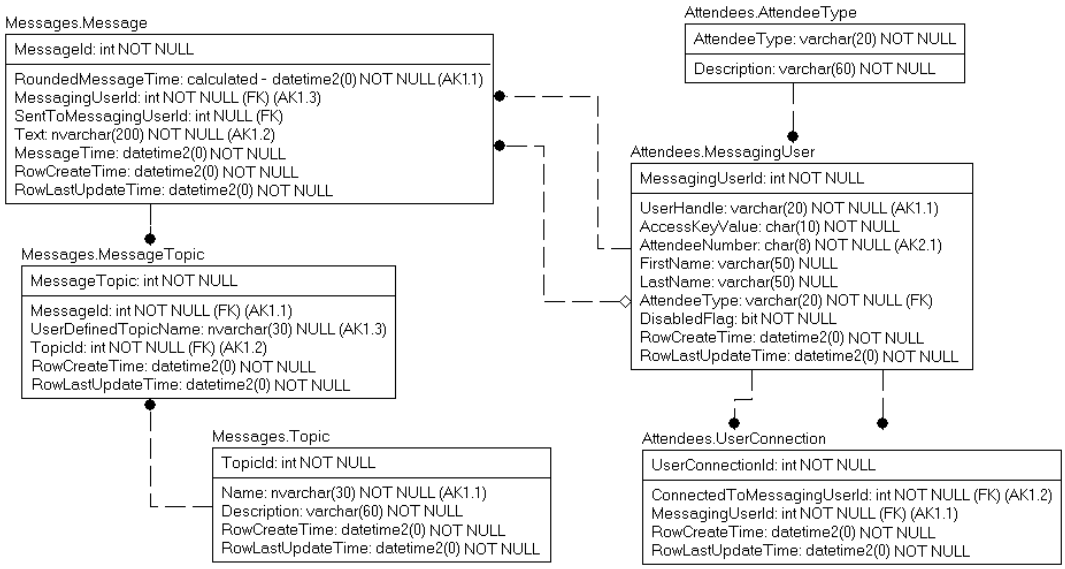


Figure 6-26. *Message model after adding RowCreateTime and RowLastUpdateTime to tables*

As a final note, it is generally best to only use these implementation columns strictly for metadata purposes. For example, consider the `Messages.Message` table. If you need to know when the message was created, you should use the `MessageTime` column as that value may represent the time when the user clicked the create button, even if it took five minutes to actually store the data. If you want to know when the row was created that represented the message, use the `RowCreateTime`. That is why I use such clunky names for the implementation column. Many tables will include the creation time, but that data may be modifiable. I don't want users changing

www.it-ebooks.info

the time when the row was created, so the name notes that the time of creation is strictly for the row, and I don't allow this column to be modified by anyone.

Sometimes, I will use these columns in concurrency control, but most of the time, I will use a `rowversion` type if the client can (and will) make use of it. Concurrency control is a very important topic that I will spend a full chapter on in in Chapter 11).

Using DDL to Create the Database

So far, we have been molding the model to make it fit our needs to implement. Columns were added, tables were added, and constraints were specified. Now, in this latter half of the chapter, we move toward the mechanical bits of the process, in that all that's left is to implement the tables we have spent so much time designing. The blueprints have been drawn up, and now, we can finally grab a hammer and start driving nails.

Just like in the rest of this book, I'll do this work manually using DDL, because it will help you understand what a tool is building for you. It's also a good exercise for any database architect or DBA to review the SQL Server syntax; I personally wouldn't suggest doing this on a database with 300 tables, but I definitely do know people who do this and wouldn't consider using a tool to create any of their database objects. On the other hand, the same data modeling tools that could be used to do the logical modeling can usually create the tables and often some of the associated code, saving your fingers from added wear and tear, plus giving you more time to help Mario save the princess who always seems to get herself captured. No matter how you do the work, you need to make sure that you end up with scripts of DDL that you or the tool uses to create objects in some manner in the file system, because they're invaluable tools for the DBA to apply changes to production, test, development, QA, or whatever environments have been set up to allow developers, users, and DBAs to coexist throughout the process.

It is not uncommon for DBAs to do all their work using scripts and never use a database design/generation tool, especially when they work for a company with smallish resources that they have already spent purchasing gold-plated laptops for all of the developers. Make sure that your scripts are in a source control system too, or at the very least backed up. Now in SQL Server 2012, we have two tools that we can work in, Management Studio and a new tool called Development Studio. Development Studio is the development-oriented tool that will replace Management Studio for developers and will introduce an offline version of creating database code. In this chapter, I am going to stick to the online paradigm where I create a database directly in the server that most readers who have any experience will understand naturally.

Before starting to build anything else, you'll need a database. I'll create this database using all default values, and my installation is very generic on my laptop. I use the Developer Edition, and I used all the default settings when installing. I hate to state the completely obvious, but you'll need to do this with an account that has rights to create a database, especially if you're doing this on a shared server, such as your company development server.

Choosing a database name is just as important as naming of other objects, and I tend to take the same sort of naming stance. Keep it as simple as possible to differentiate between all other databases, and follow the naming standards in place for your organization. I would try to be careful to try to standardize names across instances of SQL Server to allow moving of databases from server to server. In the code downloads, I will name the database `ConferenceMessaging`.

The steps I'll take along the way are as follows:

- *Creating the basic table structures*: Building the base objects with columns.
- *Adding uniqueness constraints*: Using primary and unique constraints to enforce uniqueness between rows in the table.
- *Building default constraints*: Assisting users in choosing proper values when it isn't obvious.

www.it-ebooks.info

- *Adding relationships*: Defining how tables relate to one another (foreign keys).
- *Implementing Basic Check Constraints*: Some domains need to be implemented a bit more strictly than using a simple datatype.
- *Documenting the database*: Including documentation directly in the SQL Server objects.
- *Validating the dependency information*: Using the catalog views and dynamic management views, you can validate that the objects you expect to depend on the existence of one another do, in fact, exist, keeping your database cleaner to manage.

I will use the following statement to create a small database in my directories have created:

```
CREATE DATABASE ConferenceMessaging ON
  PRIMARY ( NAME = N'ConferenceMessaging', FILENAME =
       N'C:\SQL\DATA\ConferenceMessaging.mdf' ,
    SIZE = 1024MB , MAXSIZE = 1024MB)
  LOG ON ( NAME = N'ConferenceMessaging_log', FILENAME =
       N'C:\SQL\LOG\ConferenceMessaging_log.ldf' ,
    SIZE = 100MB, MAXSIZE = 2048GB , FILEGROWTH = 100MB);
```

Or, if you want to just take the defaults, you can simply execute `CREATE DATABASE ConferenceMessaging`. In either case, you can see where the database files were placed by running the following statement (note that size is presented in 8 KB pages—more on the internal structures of the database storage in Chapter 10):

```
SELECT type_desc, size*8/1024 as [size (MB)],physical_name
FROM  sys.master_files
WHERE database_id = db_id('ConferenceMessaging');
```

This returns

```
type_desc            size (MB)            physical_name
-----------          ---------------      ------------------------------------
ROWS                 1024                 C:\SQL\DATA\ConferenceMessaging.mdf
LOG                  100                  C:\SQL\LOG\ConferenceMessaging_log.ldf
```

Next, we want to deal with the owner of the database. The database is owned by the user who created the database, as you can see from the following query (note that you should be in the context of the `ConferenceMessaging` database for this to work):

```
--determine the login that is linked to the dbo user in the database
SELECT  suser_sname(sid) as databaseOwner
FROM    sys.database_principals
WHERE   name = 'dbo';
```

On my instance, I created the database using a user named `denali` with a machine named `DENALI-PC`:

```
databaseOwner
--------------
denali-PC\denali
```


www.it-ebooks.info

You can see the owner of all databases on an instance using the following query:

```
--Get the login of owner of the database from all database
SELECT  SUSER_SNAME(owner_sid) AS databaseOwner, name
FROM    sys.databases;
```

On a typical corporate production server, I almost always will set the owner of the database to be the system administrator account so that all databases are owned by the same user. The only reason to not do this is when you are sharing databases or when you have implemented cross-database security that needs to be different for multiple databases (more information about security in Chapter 10 on security). You can change the owner of the database by using the `ALTER AUTHORIZATION` statement:

```
ALTER AUTHORIZATION ON Database::ConferenceMessaging TO SA;
```

Going back and checking the code, you will see that the owner is now `SA`.

■ **Tip** Placing semicolons at the end of every statement in your T-SQL is fast becoming a standard that will, in future versions of SQL Server, be required.

Creating the Basic Table Structures

The next step is to create the basic tables of the database. In this section, we will form `CREATE TABLE` statements to create the tables. The following is the basic syntax for the `CREATE TABLE` statement:

```
CREATE TABLE [<database>.][<schema>.]<tablename>
(
    <column specification>
);
```

If you look in Books Online, you will see a lot of additional settings that allow you to place the table on a filegroup, partition the table onto multiple filegroups, control where maximum/overflow data is placed, and so on. Where (and why) to place your data on different filegroups other than the default will be discussed in Chapter 10 on table structures and indexing. The reason for not doing this now is that it is important to segregate the tuning process from the basic data storage implementation. Once you have the database created, developers can use it to start creating code, and tuning can commence as the usage patterns emerge.

■ **Tip** Don't make this your only source of information about the DDL in SQL Server. Books Online is another great place to get exhaustive coverage of the DDL, and other sorts of books will cover the physical aspects of table creation in great detail. In this book, we focus largely on the relational aspects of database design with enough of the physical implementation to start you on the right direction. Hardware and tuning are really deep subjects that can spin in many different directions based on just how busy and large your databases are.

The base `CREATE` clause is straightforward:

```
CREATE TABLE [<database>.][<schema>.] <tablename>
```


www.it-ebooks.info

I'll expand on the items between the angle brackets (< and >). Anything in square brackets ([and]) is optional.

- <database>: It's seldom necessary to specify the database in the CREATE TABLE statements. If not specified, this defaults to the current database where the statement is being executed. Specifying the database means that the script will be executable only in a single database, which precludes us from using the script unchanged to build alternately named databases on the same server, should the need arise.
- <schema>: This is the schema to which the table will belong. We specified a schema in our model, and we will create schemas as part of this section
- <tablename>: This is the name of the table.

For the table name, if the first character of the table name is a single # symbol, the table is a temporary table. If the first two characters of the table name are ##, it's a global temporary table. Temporary tables are not so much a part of database design as a mechanism to hold intermediate results in complex queries, so don't use them in your database design. You can also create a local variable table that has the same scope as a variable by using an @ in front of the name that can be used to hold small sets of data.

The combination of schema and tablename must be unique in a database. In versions of SQL Server prior to 2005, the second part of the name was the owner, and almost every best-practice guide would suggest that all tables were owned by the dbo (database user).

Schema

As discussed in the earlier section where we defined schemas for our database, a **schema** is a namespace: a container where database objects are contained, all within the confines of a database. One thing that is nice is that because the schema isn't tightly tied to a user, you can drop the user without changing the exposed name of the object. Changing owners of the schema changes owners of the table. (This is done using the ALTER AUTHORIZATION statement.)

In SQL Server 2000 and earlier, the table was owned by a user, which made using two (or more) part names difficult. Without getting too deep into security, objects owned by the same user are easier to handle with security due to ownership chaining. If one object references another and they are owned by the same user, the ownership chain isn't broken. So we had every object owned by the same user.

Starting with SQL Server 2005, a schema is owned by a user, and tables are contained in a schema. Just as in 2000, the generally suggested best practice was that all tables were owned by the dbo user. Now, this is done by having the schema **owned** by dbo, but this doesn't mean you have to have every schema **named** dbo since an object in schema A can reference objects in schema B without breaking the ownership chain as long as they are both owned by dbo.

Not just tables are bound to a given schema; just about every object is schema bound. You can access objects using the following naming method:

```
[<databaseName>.][<schemaName>.]objectName
```

The <databaseName> defaults to the current database. The <schemaName> defaults to the user's default schema. In general, it is best to always specify the schema in any and all SQL statements because it saves SQL the work of having to decide which schema to use (when the schema is not specified, the call is considered to be *caller dependent* because what it refers to may change user to user).

www.it-ebooks.info

Schemas are of great use to segregate objects within a database for clarity of use. In our database, we have already specified two schemas, Messages and Attendees. The basic syntax is simple, just CREATE SCHEMA <schemaName> (it must be the first statement in the batch.) So I will create them using the following commands:

```
CREATE SCHEMA Messages; --tables pertaining to the messages being sent
GO
CREATE SCHEMA Attendees; --tables pertaining to the attendees and how they can send messages
GO
```

You can view the schemas created using the sys.schemas catalog view:

```
SELECT name, USER_NAME(principal_id) as principal
FROM   sys.schemas
WHERE  name <> USER_NAME(principal_id); --don't list user schemas
```

This returns

```
name                           principal
---------                      ----------
Messages                       dbo
Attendees                      dbo
```

Sometimes, schemas end up owned by a user other than dbo, like when a developer without db_owner privileges creates a schema. You can change the ownership using ALTER AUTHORIZATION, just like for the database:

```
ALTER AUTHORIZATION ON SCHEMA::Messages To DBO;
```

As a note, it is suggested to always specify the two-part name for objects in code. It is safer, because you know what schema it is using, and it doesn't need to check the default on every execution. However, for ad-hoc access, it can be annoying to type the schema if you are commonly using a certain schema. You can set a default schema for a user in the CREATE and ALTER USER statements, like this:

```
CREATE USER <schemaUser>
       FOR LOGIN <schemaUser>
       WITH DEFAULT SCHEMA = schemaname;
```

There's also an ALTER USER command that allows the changing of default schema for existing users (and in SQL Server 2012, it now works for Windows Group–based users as well. For 2005-2008R2, it only worked for standard users).

Columns and Base Datatypes

The next part of the CREATE TABLE statement is for the column specifications:

```
CREATE TABLE [<database>.][<schema>.]<tablename>
   (
      <columnName> <datatype> [<NULL specification>]
         [IDENTITY [(seed,increment)]
      --or
      <columnName> AS <computed definition>
   );
```

The <columnName> placeholder is where you specify the name of the column.

www.it-ebooks.info

There are two types of columns:

- *Implemented*: This is an ordinary column, in which physical storage is allocated and data is stored for the value.
- *Computed (or virtual)*: These columns are made up by a calculation derived from any of the physical columns in the table.

Most of the columns in any database will be implemented columns, but computed columns have some pretty cool uses, so don't think they're of no use just because they aren't talked about much. You can avoid plenty of code-based denormalizations by using computed columns. In our example tables, we specified one computed column shown in Figure 6-27.

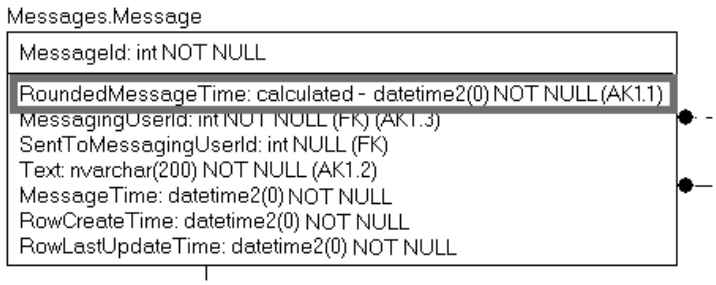


Figure 6-27. *Message table with computed column highlighted*

So the basic columns (other than the computed column) are fairly simple, just name and datatype:.

```
MessageId               int,
SentToMessagingUserId   int,
MessagingUserId         int,
Text                    nvarchar(200),
MessageTime             datetime2(0),
RowCreateTime           datetime2(0),
RowLastUpdateTime       datetime2(0)
```

The requirements called for the person to not send the same message more than once an hour. So we construct a function that takes the `MessageTime` in `datetime2(0)` datatype. That time is at a level of seconds, and we need the data in the form of hours. So first, we develop an expression that will do this. I start out with a variable of the type of the column we are deriving from and then set it to some value. I start with a variable of `datetime2(0)` and load it with the time from `SYSDATETIME()`:

```
declare @time datetime2(0)
set @time = SYSDATETIME()
```

Next, I write the following expression:

```
dateadd(hour,datepart(hour,@time),cast(cast(@time as date)as datetime2(0)) )
```

which can be broken down fairly simply, but basically takes the number of hours since midnight and adds that to the date-only value by casting it to a `date` and then to a `datetime2`, which allows you to add hours to it. Once

www.it-ebooks.info

the column is tested, you replace the variable with the MessageTime column. So, our calculated column will be specified as follows:

```
,RoundedMessageTime as dateadd(hour,datepart(hour,MessageTime),
   cast(cast(MessageTime as date)as datetime2(0)) ) PERSISTED
```

The persisted specification indicates that the value will be calculated and saved as part of the definition and storage of the table, just like a fully implemented column. In order to be persisted, the expression must be deterministic, which basically means that for the same input, you will always get the same output (much like we covered in normalization). You can also use a deterministic expression as a column in an index (we will use it as part of the uniqueness constraint for this table). So an expression like getdate() is possible, but you could not index it.

Nullability

In the column-create phrase, simply change the <NULL specification> in your physical model to NULL to allow NULLs, or NOT NULL not to allow NULLs:

```
<columnName><data type>[<NULL specification>]
```

There's nothing particularly surprising here. For the noncomputed columns in the Messages.Message table back in Figure 6-27, we will specify the following nullabilities:

```
MessageId               int             NOT NULL,
SentToMessagingUserId   int             NULL ,
MessagingUserId         int             NOT NULL ,
Text                    nvarchar(200)   NOT NULL ,
MessageTime             datetime2(0)    NOT NULL ,
RowCreateTime           datetime2(0)    NOT NULL ,
RowLastUpdateTime       datetime2(0)    NOT NULL
```

■ **Note** Leaving off the NULL specification altogether, the SQL Server default is used. To determine the current default property for a database, check the column is_ansi_null_default_on for the database in sys.databases and it can be changed using ALTER DATABASE ANSI_NULL_DEFAULT. It is so much of a best practice to ALWAYS specify the nullability of a column that I won't attempt to demonstrate how that works, as it is quite confusing.

Managing Nonnatural Primary Keys

Finally, before getting too excited and completing the table creation script, there's one more thing to discuss. Earlier in this chapter, I discussed the basics of using a surrogate key in your table. In this section, I'll present the method that I typically use. I break down surrogate key values into the types that I use:

- Manually managed, where you let the client choose the surrogate value, usually a code or short value that has meaning to the client. Sometimes, this might be a meaningless value, if the textual representation of the code needs to change frequently, but often, it will just be a simple code value. For example, if you had a table of U.S. states, you might use 'TN' for the code of the state of Tennessee.

www.it-ebooks.info

- Automatically generated, using the IDENTITY property or a GUID stored in a uniqueidentifier type column.
- A cross between the two, where you use the IDENTITY property for user created data but manually load some values to give programmers direct access to the surrogate value.

Of course, if your tables don't use any sort of surrogate values, you can move on to the next section.

Manually Managed

In the example model, I have one such situation in which I set up a domain table where I won't allow users to add or subtract rows from the table. Changes to the rows in the table could require changes to the code of the system and application layers of the application. Hence, instead of building tables that require code to manage, as well as user interfaces, we simply choose a permanent value for each of the surrogate values. This gives you control over the values in the key (which you pretty much won't have when using the IDENTITY property) and allows usage of the surrogate key directly in code if desired (likely as a constant construct in the host language). It also allows a user interface to cache values from this table or to even implement them as constants, with confidence that they won't change without the knowledge of the programmer who is using them (see Figure 6-28).

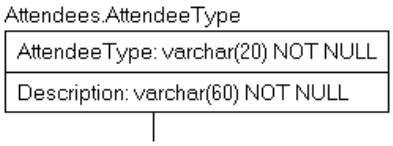


Figure 6-28. *AttendeeType table for reference*

Note that it's generally expected that once you manually create a value, the meaning of this value will never change. For example, you might have a row, `('SPEAKER', 'Persons who are speaking at the conference and have special priviledges')`. In this case, it would be fine to change the `Description` but not the value for `AttendeeType`.

Generation Using the IDENTITY Property

Most of the time, tables are created to allow users to create new rows. Implementing a surrogate key on these tables is commonly done using (what are commonly referred to as) IDENTITY columns. For any of the precise numeric datatypes, there's an option to create an automatically incrementing (or decrementing, depending on the increment value) column. The identity value increments automatically, and it works outside of transactions and locking so it works extremely fast. The column that implements this IDENTITY column should also be defined as NOT NULL. From our initial section on columns, I had this for the column specification:

```
<columnName> <data type> [<NULL specification>] IDENTITY [(seed,increment)]
```

The seed portion specifies the number that the column values will start with, and the increment is how much the next value will increase. For example, take the Movie table created earlier, this time implementing the IDENTITY-based surrogate key:

```
MessageId               int             NOT NULL IDENTITY(1,1) ,
SentToMessagingUserId   int             NULL ,
MessagingUserId         int             NOT NULL ,
```


www.it-ebooks.info

```
Text                    nvarchar(200)     NOT NULL ,
MessageTime             datetime2(0)      NOT NULL ,
RowCreateTime           datetime2(0)      NOT NULL ,
RowLastUpdateTime       datetime2(0)      NOT NULL
```

To the column declaration for the `MessageId` column of the `Message` table we have been using in the past few sections. I've added the `IDENTITY` property for the `MovieId` column. The seed of `1` indicates that the values will start at 1, and the increment says that the second value will be 1 greater, in this case 2, the next 3, and so on. You can set the seed and increment to any value that is of the datatype of the column it is being applied to. For example, you could declare the column as `IDENTITY(1000,-50)`, and the first value would be 1000, the second 950, the third 900, and so on.

The `IDENTITY` property is useful for creating a surrogate primary key that's small and fast. The `int` datatype requires only 4 bytes and is good because most tables will have fewer than 2 billion rows. There are, however, a couple of *major* caveats that you have to understand about `IDENTITY` values:

- `IDENTITY` values are apt to have holes in the sequence. If an error occurs when creating a new row, the `IDENTITY` value that was going to be used will be lost to the identity sequence. This is one of the things that allows them to be good performers when you have heavy concurrency needs. Because `IDENTITY` values aren't affected by transactions, other connections don't have to wait until another's transaction completes.
- If a row gets deleted, the deleted value won't be reused unless you insert a row yourself (not a simple task). Hence, you shouldn't use `IDENTITY` columns if you cannot accept this constraint on the values in your table.
- The value of a column with the `IDENTITY` property cannot be updated. You can insert your own value by using `SET IDENTITY_INSERT <tablename>ON`, but for the most part, you should use this only when starting a table using values from another table.
- You cannot alter a column to turn on the `IDENTITY` property, but you can add an `IDENTITY` column to an existing table.

Keep in mind the fact (I hope I've said this enough) that the surrogate key should not be the only key on the table or that the only uniqueness is a more or less random value!

Generation Using a Default Constraint

Using identity values you get a very strict key management system, where you have to use special syntax (`SET IDENTITY_INSERT`) to add a new row to the table. Instead of using a strict key generation tool like the identity, you can use a sequence object to generate new values for you. Like an identity column, it is not subject to transactions, so it is really fast, but a rollback will not recover a value that is used, leaving gaps on errors/rollbacks.

For our database, I will use the sequence object with a default constraint instead of the identity column for the key generator of the `Topic` table. Users can add new general topics, but special topics will be added manually with a specific value. I will start the user generated key values at `10000`, since it is unlikely that 10,000 specially coded topics will be needed.

```
CREATE SEQUENCE Messages.TopicIdGenerator
AS INT
MINVALUE 10000 --starting value
NO MAXVALUE --technically will max out at max int
START WITH 10000 --value where the sequence will start, differs from min based on
                 --cycle property
INCREMENT BY 1 --number that is added the previous value
NO CYCLE --if setting is cycle, when it reaches max value it starts over
```


www.it-ebooks.info

```
CACHE 100; --Use adjust number of values that SQL Server caches. Cached values would
           --be lost if the server is restarted, but keeping them in RAM makes access faster;
```

You can get the first values using the NEXT VALUE statement for sequence objects.

```
SELECT NEXT VALUE FOR Messages.TopicIdGenerator AS TopicId
UNION ALL
SELECT NEXT VALUE FOR Messages.TopicIdGenerator AS TopicId;
```

This returns

```
TopicId
--------
10000
```

■ **Note** The default datatype for a sequence object is bigint, and the default starting point is the smallest number that the sequence supports. So if you declared CREATE SEQUENCE dbo.test and fetched the first value, you would get -9223372036854775808, which is an annoying starting place for most usages. Like almost every DDL you will use in T-SQL, it is generally desirable to specify most settings, especially those that control settings that affect the way the object works for you.

You can then reset the sequence to the START WITH value using the ALTER SEQUENCE statement with a RESTART clause:

```
--To start a certain number add WITH <starting value literal>
ALTER SEQUENCE Messages.TopicIdGenerator RESTART;
```

For the Topic table, I will use the following column declaration to use the sequence object in a default. This is the first time I have use a default, so I will note that the name I gave the default object starts with a prefix of DFLT, followed by the schema name, underscore, table name, underscore, and then the column the default pertains to. This will be sufficient to keep the names unique and to identify the object in a query of the system catalog.

```
TopicId int NOT NULL CONSTRAINT DFLTMessage_Topic_TopicId
    DEFAULT(NEXT VALUE FOR Messages.TopicIdGenerator),
```

In the final section of this chapter, I will load some data for the table to give an idea of all the parts work together. One additional super-nice property of sequence objects is that you can preallocate values to allow for bulk inserts. So if you want to load 100 topic rows, you can get the values for use, build your set, and then do the insert. The allocation is done using a system stored procedure:

```
DECLARE @range_first_value sql_variant, @range_last_value sql_variant,
        @sequence_increment sql_variant;

EXEC    sp_sequence_get_range @sequence_name = N'Messages.TopicIdGenerator'
        , @range_size = 100
        , @range_first_value = @range_first_value OUTPUT
        , @range_last_value = @range_last_value OUTPUT
        , @sequence_increment = @sequence_increment OUTPUT;
```


www.it-ebooks.info

```
SELECT  CAST(@range_first_value as int) as firstTopicId,
        CAST(@range_last_value as int) as lastTopicId,
        CAST(@sequence_increment as int) as increment;
```

Since our object was just reset, the first 100 values are returned, along with the increment (something you should not assume when you use these values and you want to follow the rules of the object):

```
firstTopicId        lastTopicId       increment
------------        -----------       ----------
10000               10099             1
```

If you want to get metadata about the sequences in the database, you can use the `sys.sequences` catalog view.

```
SELECT start_value, increment, current_value
FROM   sys.sequences
WHERE  scheama_name(schema_id) = 'Messages'
AND    name = 'TopicIdGenerator';
```

For the `TopicGenerator` object we set up, this returns

```
start_value       increment       current_value
------------      -----------     --------------
10000             1               10099
```

Sequences can be a great improvement on identities, especially whenever you have any need to control the values in the surrogate key (like having unique values across multiple tables). They are a bit more work than identity values, but the flexibility is worth it when you need it. I foresee identity columns to remain the standard way of creating surrogate keys for most purposes, as their inflexibility offers some protection against having to manage data in the surrogate key, since you have to go out of your way to insert a value other than what the next identity value is with `SET IDENTITY_INSERT ON`.

■ **Tip** An alternative method for creating a surrogate key is to use GUIDs. If you use GUIDs, you would use the uniqueidentifier data type and use a default of `NewID()` or `NewSequentialId()`. Both generate new GUID values, and the `NewSequentialID` generates the GUID values to limit the amount of page splits that need to occur (page splits are a topic for Chapter 10 on data structures), which makes them less likely to be unique, but still having a duplicate would be very unlikely.

DDL to Build Tables

We have finally reached the point were we are going to create the basic table structures we have specified, including generating the primary key and the calculated column that we created. Note that we have already created the schema and sequence objects earlier in the chapter.

```
CREATE TABLE Attendees.AttendeeType (
    AttendeeType varchar(20) NOT NULL ,
    Description varchar(60) NOT NULL
);
```


www.it-ebooks.info

```
--As this is a non-editable table, we load the data here to
--start with
INSERT INTO Attendees.AttendeeType
VALUES ('Regular', 'Typical conference attendee'),
      ('Speaker', 'Person scheduled to speak'),
      ('Administrator','Manages System');

CREATE TABLE Attendees.MessagingUser (
    MessagingUserId     int NOT NULL IDENTITY ( 1,1 ) ,
    UserHandle          varchar(20) NOT NULL ,
    AccessKeyValue      char(10) NOT NULL ,
    AttendeeNumber      char(8) NOT NULL ,
    FirstName           varchar(50) NULL ,
    LastName            varchar(50) NULL ,
    AttendeeType        varchar(20) NOT NULL ,
    DisabledFlag        bit NOT NULL ,
    RowCreateTime       datetime2(0) NOT NULL ,
    RowLastUpdateTime   datetime2(0) NOT NULL
);
CREATE TABLE Attendees.UserConnection
(
    UserConnectionId    int NOT NULL IDENTITY ( 1,1 ) ,
    ConnectedToMessagingUserId int NOT NULL ,
    MessagingUserId     int NOT NULL ,
    RowCreateTime       datetime2(0) NOT NULL ,
    RowLastUpdateTime   datetime2(0) NOT NULL
);

CREATE TABLE Messages.Message (
    MessageId int NOT NULL IDENTITY ( 1,1 ) ,
    RoundedMessageTime as (dateadd(hour,datepart(hour,MessageTime),
        CAST(CAST(MessageTime as date)as datetime2(0)) ))
        PERSISTED,
    SentToMessagingUserId int NULL ,
    MessagingUserId       int NOT NULL ,
    Text                  nvarchar(200) NOT NULL ,
    MessageTime           datetime2(0) NOT NULL ,
    RowCreateTime         datetime2(0) NOT NULL ,
    RowLastUpdateTime     datetime2(0) NOT NULL
);

CREATE TABLE Messages.MessageTopic (
    MessageTopicId        int NOT NULL IDENTITY ( 1,1 ) ,
    MessageId             int NOT NULL ,
    UserDefinedTopicName  nvarchar(30) NULL ,
    TopicId               int NOT NULL ,
    RowCreateTime         datetime2(0) NOT NULL ,
    RowLastUpdateTime     datetime2(0) NOT NULL
);
```


www.it-ebooks.info

```
CREATE TABLE Messages.Topic (
    TopicId             int NOT NULL CONSTRAINT DFLTMessage_Topic_TopicId
       DEFAULT(NEXT     VALUE FOR Messages.TopicIdGenerator),
    Name                nvarchar(30) NOT NULL ,
    Description         varchar(60) NOT NULL ,
    RowCreateTime       datetime2(0) NOT NULL ,
    RowLastUpdateTime   datetime2(0) NOT NULL
);
```

After running this script, you are getting pretty far down the path, but there are still quite a few steps to go before we get finished, but sometimes, this is as far as people go when building a "small" system. It is important to do all of the steps in this chapter for almost every database you create to maintain a reasonable level of data integrity.

Adding Uniqueness Constraints

As I've mentioned several times, it's important that every table have at least one constraint that prevents duplicate rows from being created. In this section, I'll introduce the following tasks, plus a topic (indexes) that inevitably comes to mind when I start talking keys that are implemented with indexes:

- Adding primary key constraints
- Adding alternate (`UNIQUE`) key constraints
- Viewing uniqueness constraints
- Where other indexes fit in

Both types of constraints are implemented on top of unique indexes to enforce the uniqueness. It's conceivable that you could use unique indexes instead of constraints, but using a constraint is the favored method of implementing a key and enforcing uniqueness.

Constraints are intended to semantically represent and enforce constraints on data, and indexes (which are covered in detail in Chapter 9) are intended to speed access to data. In actuality, it doesn't matter how the uniqueness is implemented, but it is necessary to have either unique indexes or unique constraints in place. In some cases, other RDBMSs don't always use indexes to enforce uniqueness by default. They can use hash tables that are good only to see whether the values exist but not to look up values. By and large, when you need to enforce uniqueness, it's also the case that a user or process will be searching for values in the table and often for a single row, which indexes are perfect for.

Adding Primary Key Constraints

The first set of constraints we will add to the tables will be the primary key constraints. The syntax of the primary key declaration is straightforward:

```
[CONSTRAINT constraintname] PRIMARY KEY [CLUSTERED | NONCLUSTERED]
```

As will all constraints, the constraint name is optional, but you should never treat it as such. I'll name primary key constraints using a name such as `PK_<schema>_<tablename>`. In almost all cases, you will want to make the primary key clustered, especially when it is the most frequently used key for accessing rows.
In Chapter 10, I will describe the physical/internal structures of the database and will give more indications of when you might alter from the clustered primary key path, but I generally start with clustered and adjust if the usage patterns lead you down a different path.

www.it-ebooks.info

■ **Tip** The primary key and other constraints of the table will be members of the table's schema, so you don't need to name your constraints for uniqueness over all objects, just those in the schema.

You can specify the primary key constraint when creating the table, just like we did the default for the sequence object. If it is a single column key, you could add it to the statement like this:

```
CREATE TABLE Messages.Topic (
  TopicId int NOT NULL CONSTRAINT DFLTMessage_Topic_TopicId
    DEFAULT(NEXT VALUE FOR dbo.TopicIdGenerator)
    CONSTRAINT PK_Messages_Topic PRIMARY KEY,
  Name                nvarchar(30) NOT NULL ,
  Description         varchar(60)  NOT NULL ,
  RowCreateTime       datetime2(0) NOT NULL ,
  RowLastUpdateTime   datetime2(0) NOT NULL
);
```

Or if it was a multiple column key, you can specify it inline with the columns like the following example:

```
CREATE TABLE Examples.ExampleKey
(
  ExampleKeyColumn1 int NOT NULL,
  ExampleKeyColumn2 int NOT NULL,
  CONSTRAINT PK_Examples_ExampleKey
    PRIMARY KEY (ExampleKeyColumn1, ExampleKeyColumn2)
)
```

The more common method is use the ALTER TABLE statement and simply alter the table to add the constraint, like the following, which is the code in the downloads that will add the primary keys:

```
ALTER TABLE Attendees.AttendeeType
    ADD CONSTRAINT PK_Attendees_AttendeeType PRIMARY KEY CLUSTERED (AttendeeType);

ALTER TABLE Attendees.MessagingUser
    ADD CONSTRAINT PK_Attendees_MessagingUser PRIMARY KEY CLUSTERED (MessagingUserId);

ALTER TABLE Attendees.UserConnection
    ADD CONSTRAINT PK_Attendees_UserConnection PRIMARY KEY CLUSTERED (UserConnectionId);

ALTER TABLE Messages.Message
    ADD CONSTRAINT PK_Messages_Message PRIMARY KEY CLUSTERED (MessageId);

ALTER TABLE Messages.MessageTopic
    ADD CONSTRAINT PK_Messages_MessageTopic PRIMARY KEY CLUSTERED (MessageTopicId);

ALTER TABLE Messages.Topic
    ADD CONSTRAINT PK_Messages_Topic PRIMARY KEY CLUSTERED (TopicId);
```


www.it-ebooks.info

■ **Tip** Although the CONSTRAINT <constraintName> part of any constraint declaration isn't required, it's a very good idea always to name constraint declarations using some name. Otherwise, SQL Server will assign a name for you, and it will be ugly and will be different each and every time you execute the statement. For example, create the following object in tempdb:

```
CREATE TABLE TestConstraintName (TestConstraintNameId int PRIMARY KEY);
```

Look at the object name with this query:

```
SELECT constraint_name
FROM   information_schema.table_constraints
WHERE  table_schema = 'dbo'
 AND   table_name = 'TestConstraintName';
```

You see the name chosen is something ugly like PK__TestCons__BA850E1F645CD7F4.

Adding Alternate Key Constraints

Alternate key creation is an important task of implementation modeling. Enforcing these keys is probably more important than for primary keys, especially when using an artificial key. When implementing alternate keys, it's best to use a UNIQUE constraint. These are pretty much the same thing as primary key constraints and can even be used as the target of a relationship (relationships are covered later in the chapter).

The syntax for their creation is as follows:

```
[CONSTRAINT constraintname] UNIQUE [CLUSTERED | NONCLUSTERED] [(ColumnList)]
```

Just like the primary key, you can declare it during table creation or as an alter statement. I usually use an alter statement for code I am managing, because having the table create separate seems cleaner, but as long as the constraints get implemented, either way is fine.

```
ALTER TABLE Messages.Message
    ADD CONSTRAINT AK_Messages_Message_TimeUserAndText UNIQUE
    (RoundedMessageTime, MessagingUserId, Text);

ALTER TABLE Messages.Topic
    ADD CONSTRAINT AK_Messages_Topic_Name UNIQUE (Name);

ALTER TABLE Messages.MessageTopic
    ADD CONSTRAINT AK_Messages_MessageTopic_TopicAndMessage UNIQUE
    (MessageId, TopicId, UserDefinedTopicName);

ALTER TABLE Attendees.MessagingUser
    ADD CONSTRAINT AK_Attendees_MessagingUser_UserHandle UNIQUE (UserHandle);

ALTER TABLE Attendees.MessagingUser
    ADD CONSTRAINT AK_Attendees_MessagingUser_AttendeeNumber UNIQUE
    (AttendeeNumber);

ALTER TABLE Attendees.UserConnection
    ADD CONSTRAINT AK_Attendees_UserConnection_Users UNIQUE
    (MessagingUserId, ConnectedToMessagingUserId);
```


www.it-ebooks.info

The only really interesting tidbit here is in the `Messages.Message` declaration. Remember in the table declaration this was a computed column, so now, by adding this constraint, we have prevented the same message from being entered more than once per hour. This should show you that you can implement some fairly complex constraints using the basic building blocks we have covered so far. I will note that again that the computed column you specify must be a deterministic to be used in an index. Declaring the column as persisted is a good way to know if it is deterministic or not.

In the next few chapters, we will cover many different patterns for using these building blocks in very interesting ways. Now, we have covered all of the uniqueness constraints that were needed in our `ConferenceMessaging` database.

What About Indexes?

The topic of indexes is one that generally begins to be discussed before the first row of data is loaded into the first table. Indexes have a singular responsibility for increasing performance. At the same time, they have to be maintained, so they decrease performance too, though hopefully considerably less than they increase it. This conundrum is the foundation of the "science" of performance tuning. Hence, it is best to leave any discussion of adding indexes until data is loaded into tables and queries are executed that show the need for indexes.

In the previous section, we created uniqueness constraints whose purpose is to constrain the data in some form to make sure integrity is met. These uniqueness constraints we have just created are actually built using unique indexes and will also incur some performance penalty just like any index will. To see the indexes that have been created for your constraints, you can use the `sys.indexes` catalog view:

```
SELECT OBJECT_SCHEMA_NAME(object_id) + '.'
    + OBJECT_NAME(object_id) as object_name,
    Name,is_primary_key, is_unique_constraint
FROM sys.indexes
WHERE OBJECT_SCHEMA_NAME(object_id) <> 'sys'
    AND is_primary_key = 1 OR is_unique_constraint = 1
ORDER BY object_name;
```

which, for the constraints we have created so for, returns:

```
object_name                 Name                              primary_key  unique_constraint
--------------------------  --------------------------------  -----------  -----------------
Attendees.AttendeeType      PK_Attendees_AttendeeType         1            0
Attendees.MessagingUser     PK_Attendees_MessagingUser        1            0
Attendees.MessagingUser     AK_Attendees_MessagingUser_U…     0            1
Attendees.MessagingUser     AK_Attendees_MessagingUser_A…     0            1
Attendees.UserConnection    PK_Attendees_UserConnection       1            0
Attendees.UserConnection    AK_Attendees_UserConnection_…     0            1
Messages.Message            PK_Messages_Message               1            0
Messages.Message            AK_Messages_Message_TimeUser…     0            1
Messages.MessageTopic       PK_Messages_MessageTopic          1            0
Messages.MessageTopic       AK_Messages_MessageTopic_Top…     0            1
Messages.Topic              PK_Messages_Topic                 1            0
Messages.Topic              AK_Messages_Topic_Name            0            1
```


www.it-ebooks.info

As you start to do index tuning, one of the major tasks will be to determine whether indexes are being used and eliminate the indexes that are never (or very rarely) used to optimize queries, but you will not want to remove any indexes that show up in the results of the previous query, because they are there for data integrity purposes.

Building Default Constraints

If a user doesn't know what value to enter into a table, the value can be omitted, and the default constraint sets it to a valid predetermined value. This helps, in that you help users avoid having to make up illogical, inappropriate values if they don't know what they want to put in a column yet they need to create a row. However, the true value of defaults is lost in most applications, because the user interface would have to honor this default and not reference the column in an insert operation (or use the `DEFAULT` keyword for the column value for a default constraint to matter).

We used a default constraint earlier to implement the primary key generation, but here, I will spend a bit more time describing how it works. The basic syntax for the default constraint is

```
[CONSTRAINT constraintname] DEFAULT (<simple scalar expression>)
```

The scalar expression must be a literal, or it can use a function, even a user defined one that accesses a table. A *literal* is a simple single value in the same datatype that requires no translation by SQL Server. For example, Table 6-7 has sample literal values that can be used as defaults for a few datatypes.

Table 6-7. *Sample Default Values*

Datatype	Possible Default Value
`Int`	`1`
`varchar(10)`	`'Value'`
`binary(2)`	`0x0000`
`Datetime`	`'20080101'`

As an example in our sample database, we have the `DisabledFlag` on the `Attendees.MessagingUser` table. I'll set the default value to `0` for this column here:

```
ALTER TABLE Attendees.MessagingUser
    ADD CONSTRAINT DFAttendees_MessagingUser_DisabledFlag
    DEFAULT (0) FOR DisabledFlag;
```

Beyond literals, you can use system functions to implement default constraints. In our model, we will use a default on all of the table's `RowCreateTime` and `RowLastUpdateTime` columns. To create these constraints, I will demonstrate one of the most useful tools in a DBA's toolbox: using the system views to generate code. Since we have to do the same code over and over, I will query the metadata in the `INFORMATION_SCHEMA.COLUMN` view, and put together a query that will generate the default constraints (you will need to set your output to text and not grids in SSMS to use this code):

```
SELECT 'ALTER TABLE ' + TABLE_SCHEMA + '.' + TABLE_NAME + CHAR(13) + CHAR(10) +
       ' ADD CONSTRAINT DFLT' + TABLE_SCHEMA + '_' + TABLE_NAME + '_' +
       COLUMN_NAME + CHAR(13) + CHAR(10) +
       ' DEFAULT (SYSDATETIME()) FOR ' + COLUMN_NAME + ';'
```


www.it-ebooks.info

```
FROM   INFORMATION_SCHEMA.COLUMNS
WHERE  COLUMN_NAME IN ('RowCreateTime', 'RowLastUpdateTime')
    AND TABLE_SCHEMA IN ('Messages','Attendees')
ORDER BY TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME;
```

This code will generate the code for ten constraints:

```
ALTER TABLE Attendees.MessagingUser
    ADD CONSTRAINT DFLTAttendees_MessagingUser_RowCreateTime
    DEFAULT (SYSDATETIME()) FOR RowCreateTime;

ALTER TABLE Attendees.MessagingUser
    ADD CONSTRAINT DFLTAttendees_MessagingUser_RowLastUpdateTime
    DEFAULT (SYSDATETIME()) FOR RowLastUpdateTime;

ALTER TABLE Attendees.UserConnection
    ADD CONSTRAINT DFLTAttendees_UserConnection_RowCreateTime
    DEFAULT (SYSDATETIME()) FOR RowCreateTime;

ALTER TABLE Attendees.UserConnection
    ADD CONSTRAINT DFLTAttendees_UserConnection_RowLastUpdateTime
    DEFAULT (SYSDATETIME()) FOR RowLastUpdateTime;

ALTER TABLE Messages.Message
    ADD CONSTRAINT DFLTMessages_Message_RowCreateTime
    DEFAULT (SYSDATETIME()) FOR RowCreateTime;

ALTER TABLE Messages.Message
    ADD CONSTRAINT DFLTMessages_Message_RowLastUpdateTime
    DEFAULT (SYSDATETIME()) FOR RowLastUpdateTime;

ALTER TABLE Messages.MessageTopic
    ADD CONSTRAINT DFLTMessages_MessageTopic_RowCreateTime
    DEFAULT (SYSDATETIME()) FOR RowCreateTime;

ALTER TABLE Messages.MessageTopic
    ADD CONSTRAINT DFLTMessages_MessageTopic_RowLastUpdateTime
    DEFAULT (SYSDATETIME()) FOR RowLastUpdateTime;

ALTER TABLE Messages.Topic
    ADD CONSTRAINT DFLTMessages_Topic_RowCreateTime
    DEFAULT (SYSDATETIME()) FOR RowCreateTime;

ALTER TABLE Messages.Topic
    ADD CONSTRAINT DFLTMessages_Topic_RowLastUpdateTime
    DEFAULT (SYSDATETIME()) FOR RowLastUpdateTime;
```

Obviously it's not the point of this section, but generating code with the system metadata is a very useful skill to have, particularly when you need to add some type of code over and over.

Adding Relationships (Foreign Keys)

I've covered relationships in previous chapters already, so I'll try to avoid saying too much more about why to use them. In this section, I'll simply discuss how to implement relationships. It's common to add constraints using the `ALTER TABLE` statement, but you can also do this using the `CREATE TABLE` statement. However, because tables

www.it-ebooks.info

are frequently created all at once, it's usually more likely that the ALTER TABLE command will be used, because parent tables needn't be created before dependent child tables in scripts.

The typical foreign key is implemented as a primary key of one table migrated to the child table that represents the entity from which it comes. You can also reference a unique constraint as well, but it is pretty rare and a very atypical implementation. An example is a table with an identity key and a textual code. You could migrate the textual code to a table to make it easier to read, if user requirements required it and you failed to win the argument against doing something that will confuse everyone for years to come.

The syntax of the statement for adding foreign key constraints is pretty simple:

```
[CONSTRAINT <constraintName>]
   FOREIGN KEY REFERENCES <referenceTable> (<referenceColumns>)
   [ON DELETE <NO ACTION | CASCADE | SET NULL | SET DEFAULT> ]
   [ON UPDATE <NO ACTION | CASCADE | SET NULL | SET DEFAULT> ]
```

where

- <referenceTable> is the parent table in the relationship.
- <referenceColumns> is a comma-delimited list of columns in the child table in the same order as the columns in the primary key of the parent table.
- ON DELETE or ON UPDATE clauses specify what to do when a row is deleted or updated. Options are
  - NO ACTION: Raises an error if you end up with a child with no matching parent after the statement completes
  - CASCADE: Applies the action on the parent to the child, either updates the migrated key values in the child to the new parent key value or deletes the child row
  - SET NULL: If you delete or change the value of the parent, you set the child key to NULL
  - SET DEFAULT: If you delete or change the value of the parent, the child key is set to the default value from the default constraint, or NULL if no constraint exists.

If you are using surrogate keys, you will very rarely need either of the ON UPDATE options, since the value of a surrogate is rarely editable. For deletes, 98 % of the time you will use NO ACTION, because most of the time, you will simply want to make the user delete the children first to avoid accidentally deleting a lot of data. Lots of NO ACTION foreign key constraints will tend to make it much harder to execute an accidental DELETE FROM <tableName> when you accidentally didn't highlight the WHERE clause in SSMS. The most useful of the actions is ON DELETE CASCADE, which is frequently useful for table sets where the child table is, in essence, just a part of the parent table. For example, invoice ←invoiceLineItem. Usually, if you are going to delete the invoice, you are doing so because it is bad ,and you will want the line items to go away too. On the other hand, you want to avoid if for relationships like Customer ←Invoice. Deleting a customer who has invoices is probably a bad idea.

Thinking back to our moeling, there were optional and required relationships such as the one in Figure 6-29.



Figure 6-29. *Optional parent-to-child relationship requires NULL on the migrated key*

www.it-ebooks.info

The `child.parentId` column needs to allow `NULL`s (which it does on the model). For a required relationship, the `child.parentId` would not be null, like in Figure 6-30.

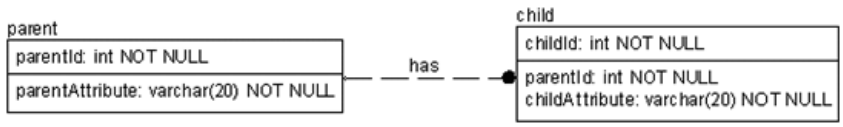


Figure 6-30. *Required parent-to-child relationship requires NOT NULL on the migrated key*

This is all you need to do, because SQL Server knows that when the referencing key allows a `NULL`, the relationship value is optional. You don't need to have a `NULL` primary key value for the relationship because, as discussed, it's impossible to have a `NULL` attribute in a primary key. In our model, represented here in Figure 6-31, we have seven relationship modeled.

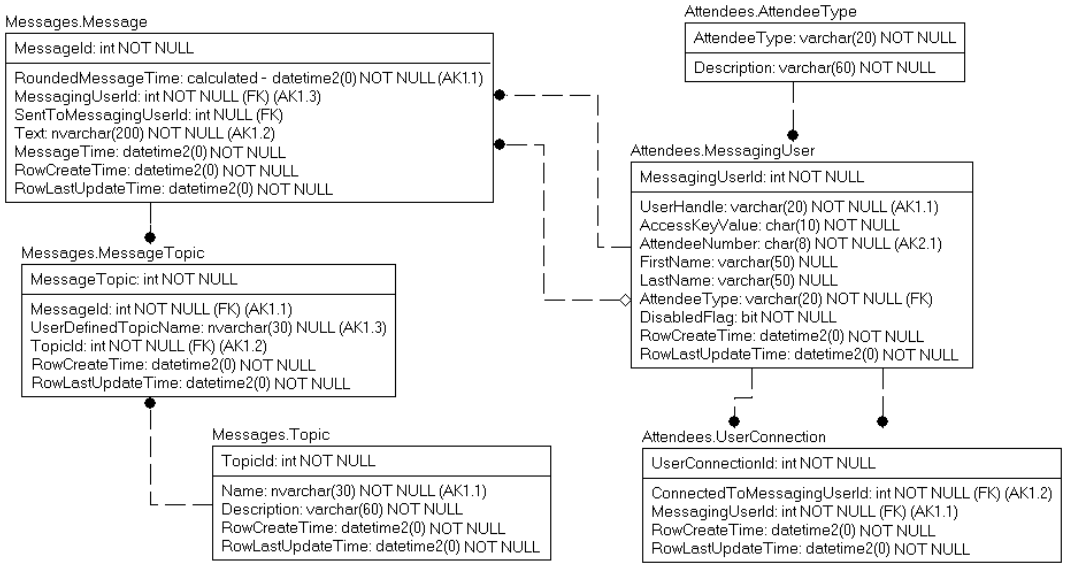


Figure 6-31. *Messaging model for reference*

In Figure 6-9, you will remember that we had given the relationships a verb phrase, which is used to read the name. For example, in the relationship between `User` and `Message`, we have two relationships. One of them was verb phrased as `"Is Sent"` as in `User-Is Sent-Message`. In order to get interesting usage of these verb phrases, I will use them as part of the name of the constraint, so that constraint will be named:

```
FK__Attendees_MessagingUser$IsSent$Messages_Message
```

By doing this, it greatly improves the value of the names for constraints, particularly when you have more than one foreign key going between the same two tables. Now, let's go through the seven constraints and decide the type of options to use on the foreign key relationships. First up is the relationship between `AttendeeType`

www.it-ebooks.info

and MessagingUser. Since it uses a natural key, it is a target for the UPDATE CASCADE option. However, it should be noted that if you have a lot of MessagingUser rows, this operation can be a very costly, so it should be done during off hours. And, if it turns out it is done very often, the choice to use a volatile natural key value ought to be reconsidered. We will use ON DELETE NO ACTION, because we don't usually want to cascade a delete from a table that is strictly there to implement a domain.

```
ALTER TABLE Attendees.MessagingUser
    ADD CONSTRAINT
    FK__Attendees_MessagingUser$IsSent$Messages_Message
        FOREIGN KEY (AttendeeType) REFERENCES Attendees.AttendeeType(AttendeeType)
        ON UPDATE CASCADE
        ON DELETE NO ACTION;
```

Next, let's consider the two relationships between the MessagingUser and the UserConnection table. Since we modeled both of the relationships as required, if one user is deleted (as opposed to being disabled), then we would delete all connections to and from the MessagingUser. Hence, you might consider implement both of these as DELETE CASCADE. However, if you execute the following statements:

```
ALTER TABLE Attendees.UserConnection
    ADD CONSTRAINT
    FK__Attendees_MessagingUser$ConnectsToUserVia$Attendees_UserConnection
    FOREIGN KEY (MessagingUserId) REFERENCES Attendees.MessagingUser(MessagingUserId)
    ON UPDATE NO ACTION
    ON DELETE CASCADE;

ALTER TABLE Attendees.UserConnection
    ADD CONSTRAINT
    FK__Attendees_MessagingUser$IsConnectedToUserVia$Attendees_UserConnection
    FOREIGN KEY (ConnectedToMessagingUserId)
        REFERENCES Attendees.MessagingUser(MessagingUserId)
    ON UPDATE NO ACTION
    ON DELETE CASCADE;
```

```
Introducing FOREIGN KEY constraint
'FK__Attendees_MessagingUser$IsConnectedToUserVia$Attendees_UserConnection' on table
'UserConnection' may cause cycles or multiple cascade paths. Specify ON DELETE NO ACTION or ON
UPDATE NO ACTION, or modify other FOREIGN KEY constraints.
```

Basically, this is stating that you cannot have two CASCADE operations on the same table. This even more limits the value of the CASCADE operations. Instead, we will use NO ACTION for the DELETE and will just have to implement the cascade in the client code or using a trigger. I will also note that, in many ways, this is probably a good thing. Too much automatically executing code is going to make developers antsy about what is going on with the data, and if you accidentally delete a user, having NO ACTION specified can actually be a good thing to stop dumb mistakes. So I will change the constraints to NO ACTION and re-create (dropping the one that was created first):

```
ALTER TABLE Attendees.UserConnection
    DROP CONSTRAINT
    FK__Attendees_MessagingUser$ConnectsToUserVia$Attendees_UserConnection
GO

ALTER TABLE Attendees.UserConnection
    ADD CONSTRAINT
```


www.it-ebooks.info

```
    FK__Attendees_MessagingUser$ConnectsToUserVia$Attendees_UserConnection
    FOREIGN KEY (MessagingUserId) REFERENCES Attendees.MessagingUser(MessagingUserId)
    ON UPDATE NO ACTION
    ON DELETE NO ACTION;

ALTER TABLE Attendees.UserConnection
    ADD CONSTRAINT
    FK__Attendees_MessagingUser$IsConnectedToUserVia$Attendees_UserConnection
    FOREIGN KEY (ConnectedToMessagingUserId)
        REFERENCES Attendees.MessagingUser(MessagingUserId)
    ON UPDATE NO ACTION
    ON DELETE NO ACTION;
Go
```

For the two relationships between `MessagingUser` and `Message`, you again might want to consider using `ON DELETE CASCADE` because if you delete a user, we will again want to get rid of all of that user's messages. Note that we would do this here because we implemented a disabled indicator, and if the user really needed to be deleted, it is very likely that all of the messages would need to be deleted. If a reasonable user was disabled for quitting the service or repeatedly using the wrong access key, you wouldn't delete them, and you would want to keep their messages. However, for the same reasons as for the previous constraints, you will have to do it manually if you decide to delete all of the messages, and that will help to give the user a choice if they have a lot of messages and maybe just need to be disabled.

```
ALTER TABLE Messages.Message
    ADD CONSTRAINT FK__Messages_MessagingUser$Sends$Messages_Message
    FOREIGN KEY (MessagingUserId)
        REFERENCES Attendees.MessagingUser(MessagingUserId)
    ON UPDATE NO ACTION
    ON DELETE NO ACTION;

ALTER TABLE Messages.Message
    ADD CONSTRAINT FK__Messages_MessagingUser$IsSent$Messages
    FOREIGN KEY (SentToMessagingUserId)
        REFERENCES Attendees.MessagingUser(MessagingUserId)
    ON UPDATE NO ACTION
    ON DELETE NO ACTION;
```

The next relationship we will deal with is between `Topic` and `MessageTopic`. We don't want `Topics` to be deleted once set up to be used, other than by the administrator as a special operation perhaps, where special requirements are drawn up and not done as a normal thing. Hence, we use the `DELETE NO ACTION`.

```
ALTER TABLE Messages.MessageTopic
    ADD CONSTRAINT FK__Messages_Topic$CategorizesMessagesVia$Messages_MessageTopic
    FOREIGN KEY (TopicId)
        REFERENCES Messages.Topic(TopicId)
    ON UPDATE NO ACTION
    ON DELETE NO ACTION;
```

The next to the last relationship to implement is the `MessageTopic` to `Message` relationship. Just like the `Topic` to `Message` topic relationship, there is no need to automatically delete messages if the topic is deleted.

```
ALTER TABLE Messages.MessageTopic
    ADD CONSTRAINT FK_Message$iscCategorizedVia$MessageTopic
    FOREIGN KEY (MessageId)
        REFERENCES Messages.Message(MessageId)
```


www.it-ebooks.info

```
    ON UPDATE NO ACTION
    ON DELETE NO ACTION;
```

One of the primary limitations on constraint-based foreign keys is that the tables participating in the relationship cannot span different databases. When this situation occurs, these relationship types need to be implemented via triggers.

It's generally a bad idea to design databases with cross-database relationships. A database should be considered a unit of related tables that are always kept in sync. When designing solutions that extend over different databases or even servers, carefully consider how spreading around references to data that isn't within the scope of the database will affect your solution. You need to understand that SQL Server cannot guarantee the existence of the value, because SQL Server uses databases as its "container," and another user could restore a database with improper values, even an empty database, and the cross-database RI would be invalidated. Of course, as is almost always the case with anything that isn't best-practice material, there are times when cross-database relationships are unavoidable, and I'll demonstrate building triggers to support this need in the next chapter on data protection.

In the security chapter (Chapter 9), we will discuss more about how to secure cross database access, but it is generally considered a less than optimal usage. In SQL Server 2012, the concepts of contained databases, and even SQL Azure, the ability to cross database boundaries is changing in ways that will generally be helpful for building secure databases that exist on the same server.

Adding Basic Check Constraints

In our database, we have specified a couple of domains that need to be implemented a bit more strictly. In most cases, we can implement validation routines using simple check constraints. Check constraints are simple, single-row predicates that can be used to validate the data in a row. The basic syntax is

```
ALTER TABLE <tableName> [WITH CHECK | WITH NOCHECK]
    ADD [CONSTRAINT <constraintName>]
    CHECK <BooleanExpression>
```

There are two interesting parts of check constraints that are different from what you have seen in previous constraints, and we need to cover these briefly before we start creating them. The first is `<BooleanExpression>`. The `<BooleanExpression>` component is similar to the `WHERE` clause of a typical `SELECT` statement, but with the caveat that no subqueries are allowed. (Subqueries are allowed in standard SQL but not in T-SQL). In T-SQL, you must use a function to access other tables, something I will use later in this book as I create more interesting check constraints to implement data protection patterns of Chapter 8.)

`CHECK` constraints can reference system and user-defined functions and use the name or names of any columns in the table. However, they cannot access any other table, and they cannot access any row other than the current row being modified (except through a function, and the row values you will be checking will already exist in the table). If multiple rows are modified, each row is checked against this expression individually.

The interesting thing about this expression is that unlike a `WHERE` clause, the condition is checked for falseness rather than truth. Without going completely into a discussion of `NULL`, it's important to understand that `CHECK` constraints fail only on rows that are explicitly `False`. If the result of a comparison is `UNKNOWN` because of a `NULL` comparison, the row will pass the check constraint and be entered.

Even if this isn't immediately confusing, it is often confusing when figuring out why an operation on a row did or did not work as you might have expected. For example, consider the Boolean expression `value <> 'fred'`. If `value` is `NULL`, this is accepted, because `NULL <> 'fred'` evaluates to `UNKNOWN`. If value is `'fred'`, it fails because `'fred' <> 'fred'` is `False`. The reason for the way `CHECK` constraints work with Booleans is that if the column is defined as `NULL`, it is assumed that you wish to allow a NULL value for the column value. You can look for `NULL` values by explicitly checking for them using `IS NULL` or `IS NOT NULL`. This is useful when you want to ensure that a column that technically allows nulls does not allow `NULL`s if another column has a given value.

www.it-ebooks.info

As an example, if you have a column defined `name varchar(10) null`, having a check constraint that says `name = 'fred'` technically says `name = 'fred' OR name IS NULL`. If you want to ensure it is not null if the column `NameIsNotNullFlag = 1`, you would state `(NameIsNotNullFlag = 1 AND Name IS NOT NULL) or (Name = 0)`.

The second interesting part of the statement is the `[WITH CHECK | WITH NOCHECK]` specification. When you create a `CHECK` constraint, the `WITH CHECK` setting (the default) gives you the opportunity to create the constraint without checking existing data. Using `NOCHECK` and leaving the values unchecked is a pretty bad thing to do, in my opinion. First off, when you try to resave the exact same data, you will get an error. Plus, if the constraint is built `WITH CHECK`, the optimizer could possibly make use of this fact when building plans if the constraint didn't use any functions and just used simple comparisons such as less than, greater than, and so on. For example, imagine you have a constraint that says that a value must be less than `10`. If, in a query, you look for all values `11` and greater, the optimizer can use this fact and immediately return zero rows, rather than having to scan the table to see whether any value matches.

In our model, we had two domains specified in the text that we will implement here. The first is the `TopicName`, which called for us to make sure that the value is not an empty string or all space characters. I repeat it here in Table 6-8 for review.

Table 6-8. *Domain: TopicName*

Property	Setting
Name	TopicName
Optional	No
Datatype	Unicode text, 30 characters
Value Limitations	Must not be an empty string or only space characters
Default Value	n/a

The maximum length of 30 characters was handled by the datatype of `nvarchar(30)` we used but now will implement the rest of the value limitations. The method I will use for this is to do a `ltrim` on the value and then check the length. If it is `0`, it is either all spaces or empty. We used the `topicName` domain for two columns, the `name` column from `Messages.Topic`, and the `UserDefinedTopicName` from the `Messages.MessageTopic` table:

```
ALTER TABLE Messages.Topic
    ADD CONSTRAINT CHK__Messages_Topic_Name_NotEmpty
      CHECK (LEN(RTRIM(Name)) > 0);

ALTER TABLE Messages.MessageTopic
    ADD CONSTRAINT CHK__Messages_MessageTopic_UserDefinedTopicName_NotEmpty
      CHECK (LEN(RTRIM(UserDefinedTopicName)) > 0);
```

The other domain we specifically mentioned was for the `UserHandle`, as repeated in Table 6-9.

Table 6-9. *Domain: UserHandle*

Property	Setting
Name	UserHandle
Optional	No
Datatype	Basic character set, maximum of 20 characters
Value Limitations	Must be 5–20 simple alphanumeric characters and start with a letter
Default Value	n/a

www.it-ebooks.info

To implement this domain, we will get a bit more interesting.

```
ALTER TABLE Attendees.MessagingUser
    ADD CONSTRAINT CHK__Attendees_MessagingUser_UserHandle_LenthAndStart
      CHECK (LEN(Rtrim(UserHandle)) >= 5
          AND LTRIM(UserHandle) LIKE '[a-z]' +  REPLICATE('[a-z1-9]',LEN(RTRIM(UserHandle)) -1));
```

The first part of the CHECK constraint Boolean expression simply checks to see if the string is greater than five characters long. The latter part creates a like expression that checks that the name starts with a letter and that the following characters are only alphanumeric. It looks like it might be slow, based on the way we are taught to write WHERE clause expressions, but in this case, you aren't searching but are working on a single row already in memory.

Finally, we had one other predicate that we need to implement. Back in the requirements, it was specified that the MessageTopic table, we need to make sure that the UserDefinedTopicName is NULL unless the Topic that is chosen is the one set up for the UserDefined topic. So we will create a new row. Since the surrogate key of MessageTopic is a default constraint using a sequence, we can simply enter the row specifying the TopicId as 0:

```
INSERT INTO Messages.Topic(TopicId, Name, Description)
VALUES (0,'User Defined','User Enters Their Own User Defined Topic');
```

Then, we add the constraint, checking to make sure that the UserDefinedTopicId is null if the TopicId = 0 and vice versa.

```
ALTER TABLE Messages.MessageTopic
    ADD CONSTRAINT CHK__Messages_MessageTopic_UserDefinedTopicName_NullUnlessUserDefined
    CHECK ((UserDefinedTopicName IS NULL AND TopicId <> 0)
      OR (TopicId = 0 AND UserDefinedTopicName IS NOT NULL));
```

Be sure to be as specific as possible with your check criteria, as it will make implementation a lot safer. Now, we have implemented all of the check constraints we are going to for our demonstration database. In the testing section later in this chapter, one of the most important things to test are the check constraints (and if you have done any advanced data integrity work in triggers, which we will leave to later chapters).

Triggers to Maintain Automatic Values

For all of our tables, we included two columns that we are going to implement as automatically maintained columns. These columns are the RowCreateTime and RowLastUpdateTime that we added earlier in this chapter (shown in Figure 6-27). These columns are useful to help us get an idea of some of the actions that have occurred on our row without resorting to looking through change tracking. Sometimes, these values have meaning to the end users as well, but most often, we are implementing them strictly for to software's sake, hence the reason that we will implement them in such a manner that the client cannot modify the values.

I will do this with an "instead of" trigger, which will, for the most part, be a very smooth way to manage automatic operations on the base table, but it does have a few downsides.

- SCOPE_IDENTITY() will no longer return the identity value that is returned, since the actual insert will be done in the trigger, outside of the scope of the code. @@identity will work, but it is own issues, particularly with triggers that perform cascading operations.
- The output clause will not work if you have triggers on the table.

The SCOPE_IDENTITY issue can be gotten around by using an AFTER trigger for an insert (which I will include as a sample in this section). I personally suggest that you ought to use one of the natural keys you have implemented to get the inserted value if you are inserting a single row anyhow.

www.it-ebooks.info

One of the downsides of triggers can be performance, so sometimes automatically generated values will simply be maintained by the SQL code that uses the tables or perhaps the columns are simply removed. I far prefer a server-based solution, because clock synchronization can be an issue when even two distinct servers are involved with keeping time. So if an action says it occurred at 12:00 AM by the table, you look in the log and at 12:00 AM, everything looks fine, but at 11:50 PM there was a glitch of some sort. Are they related? It is not possible to know to the degree you might desire.

As it is my favored mechanism for maintaining automatically maintained columns, I will implement triggers for tables, other than `Attendees.AttendeeType`, because, you should recall, we will not enable end users to make changes to the data, so tracking changes will not be needed.

To build the triggers, I will use the trigger templates that are included in Appendix B as the basis for the trigger. If you want to know more about the basics of triggers and how these templates are constructed, check Appendix B. The basics of how the triggers work should be very self explanatory. The code added to the base trigger template from the appendix will be highlighted in bold.

In the following "instead of" insert trigger, we will replicate the operation of insert on the table, passing through the values from the user insert operation, but replacing the `RowCreateTime` and `RowLastUpdateTime` with the function `SYSDATETIME()`. One quick topic we should hit on here is multirow operations. Well written triggers take into consideration that any `INSERT`, `UPDATE`, or `DELETE` operation must support multiple rows being operated on simultaneously. The `inserted` and `deleted` virtual tables house the rows that have been inserted or deleted in the operation. (For an update, think of rows as being deleted and then inserted, at least logically.)

```
CREATE TRIGGER MessageTopic$InsteadOfInsertTrigger
ON Messages.MessageTopic
INSTEAD OF INSERT AS
BEGIN

  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
    @rowsAffected int = (SELECT COUNT(*) FROM inserted)

  --@rowsAffected = (SELECT COUNT(*) FROM deleted)
  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  BEGIN TRY
    --[validation section]
    --[modification section]
    --<perform action>
    INSERT INTO Messages.MessageTopic (MessageId, UserDefinedTopicName,
        TopicId,RowCreateTime,RowLastUpdateTime)
    SELECT MessageId, UserDefinedTopicName, TopicId, SYSDATETIME(), SYSDATETIME()
    FROM inserted ;
  END TRY
  BEGIN CATCH
    IF @@trancount > 0
       ROLLBACK TRANSACTION;
    THROW; --will halt the batch or be caught by the caller's catch block
```


www.it-ebooks.info

```
  END CATCH
END
GO
```

For the update operation, we will do very much the same thing, only when we replicate the update operation, we will make sure that the RowCreateTime stays the same, no matter what the user might send in the update, and the RowLastUpdateTIme will be replaced by SYSDATETIME().

```
CREATE TRIGGER Messages.MessageTopic$InsteadOfUpdateTrigger
ON Messages.MessageTopic
INSTEAD OF UPDATE AS
BEGIN
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  @rowsAffected int = (SELECT COUNT(*) FROM inserted)
  --@rowsAffected = (SELECT COUNT(*) FROM deleted)

  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  BEGIN TRY
    --[validation section]
    --[modification section]
    --<perform action>
  UPDATE MessageTopic
  SET MessageId = Inserted.MessageId,
    UserDefinedTopicName = Inserted.UserDefinedTopicName,
    TopicId = Inserted.TopicId,
    RowCreateTime = MessageTopic.RowCreateTime, --no changes allowed
    RowLastUpdateTime = SYSDATETIME()
  FROM inserted
    JOIN Messages.MessageTopic
    ON inserted.MessageTopicId = MessageTopic.MessageTopicId;
  END TRY
  BEGIN CATCH
    IF @@trancount > 0
       ROLLBACK TRANSACTION;
    THROW; --will halt the batch or be caught by the caller's catch block
  END CATCH
END
```

If you find that using an "instead of" insert trigger is too invasive of a technique, you can change to using an after trigger. For an after trigger, you only need to update the columns that are important. It is a bit slower because it is updating the row after it is in the table, but it does work quite well. Another reason why an "instead of" trigger

www.it-ebooks.info

may not be allowed is if you have a cascade operation. For example, consider our relationship from Messging User to Attendee Type:

```
ALTER TABLE [Attendees].[MessagingUser]
  ADD CONSTRAINT [FK__Attendees_MessagingUser$IsSent$Messages_Message]
      FOREIGN KEY([AttendeeType])
      REFERENCES [Attendees].[AttendeeType] ([AttendeeType])
ON UPDATE CASCADE;
```

Since this is cascade, we will have to use an after trigger for the UPDATE trigger, since when the cascade occurs in the base table, the automatic operation won't use the trigger, but only the base table operations. So we will implement the update trigger as:

```
CREATE TRIGGER MessageTopic$UpdateRowControlsTrigger
ON Messages.MessageTopic
AFTER UPDATE AS
BEGIN

  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
    @rowsAffected int = (SELECT COUNT(*) FROM inserted)
  --@rowsAffected = (SELECT COUNT(*) FROM deleted)
  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  BEGIN TRY
      --[validation section]
      --[modification section]
  UPDATE MessageTopic
  SET    RowCreateTime = SYSDATETIME(),
         RowLastUpdateTime = SYSDATETIME()
  FROM   inserted
    JOIN Messages.MessageTopic
    on inserted.MessageTopicId = MessageTopic.MessageTopicId;
  END TRY
  BEGIN CATCH
      IF @@trancount > 0
         ROLLBACK TRANSACTION;

      THROW --will halt the batch or be caught by the caller's catch block;
  END CATCH
END
```

In the downloads for this chapter, I will include "instead of" triggers for all of the tables in our database. They will all follow the same pattern, and because of this, I will almost always use some form of code generation tool to create these triggers. We discussed code generation earlier for building default constraints for these same columns, and you could do the very same thing for building triggers. I generally use a third-party tool to do code generation, but it is essential to the learning process that you code the first ones yourself so you know how things work.

www.it-ebooks.info

Documenting Your Database

In your modeling, you've created descriptions, notes, and various pieces of data that will be extremely useful in helping the developer understand the whys and wherefores of using the tables you've created. In previous versions of SQL Server, it was difficult to make any use of this data directly in the server. In SQL Server 2000, Microsoft introduced extended properties that allow you to store specific information about objects. This is great, because it allows you to extend the metadata of your tables in ways that can be used by your applications using simple SQL statements.

By creating these properties, you can build a repository of information that the application developers can use to do the following:

- Understand what the data in the columns is used for
- Store information to use in applications, such as the following:
  - Captions to show on a form when a column is displayed
  - Error messages to display when a constraint is violated
  - Formatting rules for displaying or entering data
  - Domain information, like the domain you have chosen for the column during design

To maintain extended properties, you're given the following functions and stored procedures:

- `sys.sp_addextendedproperty`: Used to add a new extended property
- `sys.sp_dropextendedproperty`: Used to delete an existing extended property
- `sys.sp_updateextendedproperty`: Used to modify an existing extended property
- `fn_listextendedproperty`: A system-defined function that can be used to list extended properties
- `sys.extendedproperties`: Can be used to list all extended properties in a database, less friendly than `fn_listextendedproperty`

Each (other than `sys.extendedproperties`) has the following parameters:

- `@name`: The name of the user-defined property.
- `@value`: What to set the value to when creating or modifying a property.
- `@level0type`: Top-level object type, often schema, especially for most objects that users will use (tables, procedures, and so on).
- `@level0name`: The name of the object of the type that's identified in the `@level0type` parameter.
- `@level1type`: The name of the type of object such as `Table`, `View`, and so on.
- `@level1name`: The name of the object of the type that's identified in the `@level1type` parameter.
- `@level2type`: The name of the type of object that's on the level 2 branch of the tree under the value in the `@level1Type` value. For example, if `@level1type` is `Table`, then `@level2type` might be `Column`, `Index`, `Constraint`, or `Trigger`.
- `@level2name`: The name of the object of the type that's identified in the `@level2type` parameter.

www.it-ebooks.info

For our example, let's use the `Messages.Topic` table, which was defined by the following DDL:

```
CREATE TABLE Messages.Topic (
  TopicId int NOT NULL CONSTRAINT DFLTMessage_Topic_TopicId
       DEFAULT(NEXT VALUE FOR dbo.TopicIdGenerator),
  Name                nvarchar(30) NOT NULL ,
  Description         varchar(60) NOT NULL ,
  RowCreateTime       datetime2(0) NULL ,
  RowLastUpdateTime   datetime2(0) NULL
);
```

For simplicity sake, I will just be adding a property with a description of the table, but you can add whatever bits of information you may want to enhance the schema, both in usage and for management tasks. For example, you might add an extended property to tell the reindexing schemes when or how to reindex a table's indexes. To document this table, let's add a property to the table and columns named `Description`. You execute the following script after creating the table (note that I used the descriptions as outlined in the start of the chapter for the objects):

```
--Messages schema
EXEC sp_addextendedproperty @name = 'Description',
  @value = 'Messaging objects',
  @level0type = 'Schema', @level0name = 'Messages';

--Messages.Topic table
EXEC sp_addextendedproperty @name = 'Description',
  @value = ' Pre-defined topics for messages',
  @level0type = 'Schema', @level0name = 'Messages',
  @level1type = 'Table', @level1name = 'Topic';

--Messages.Topic.TopicId
EXEC sp_addextendedproperty @name = 'Description',
  @value = 'Surrogate key representing a Topic',
  @level0type = 'Schema', @level0name = 'Messages',
  @level1type = 'Table', @level1name = 'Topic',
  @level2type = 'Column', @level2name = 'TopicId';

--Messages.Topic.Name
EXEC sp_addextendedproperty @name = 'Description',
  @value = 'The name of the topic',
  @level0type = 'Schema', @level0name = 'Messages',
  @level1type = 'Table', @level1name = 'Topic',
  @level2type = 'Column', @level2name = 'Name';

--Messages.Topic.Description
EXEC sp_addextendedproperty @name = 'Description',
  @value = 'Description of the purpose and utilization of the topics',
  @level0type = 'Schema', @level0name = 'Messages',
  @level1type = 'Table', @level1name = 'Topic',
  @level2type = 'Column', @level2name = 'Description';

--Messages.Topic.RowCreateTime
EXEC sp_addextendedproperty @name = 'Description',
  @value = 'Time when the row was created',
  @level0type = 'Schema', @level0name = 'Messages',
```


www.it-ebooks.info

```
  @level1type = 'Table', @level1name = 'Topic',
  @level2type = 'Column', @level2name = 'RowCreateTime';

--Messages.Topic.RowLastUpdateTime
EXEC sp_addextendedproperty @name = 'Description',
  @value = 'Time when the row was last updated',
  @level0type = 'Schema', @level0name = 'Messages',
  @level1type = 'Table', @level1name = 'Topic',
  @level2type = 'Column', @level2name = 'RowLastUpdateTime';
```

Now, when you go into Management Studio, right-click the `Messages.Topic` table, and select Properties. Choose Extended Properties, and you see your description, as shown in Figure 6-32.

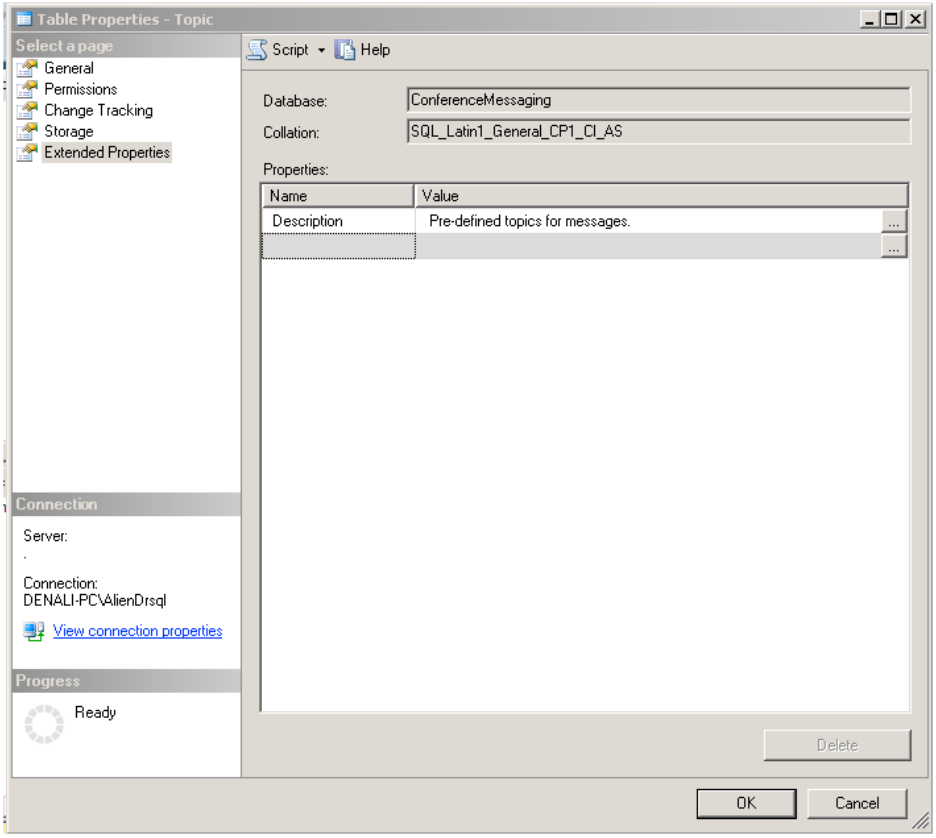


Figure 6-32. *Reward for hard work done. Descriptions in Management Studio.*

The `fn_listExtendedProperty` object is a system-defined function you can use to fetch the extended properties (the parameters are as discussed earlier—the name of the property and then each level of the hierarchy):

```
SELECT objname, value
```


www.it-ebooks.info

```
FROM fn_listExtendedProperty ( 'Description',
                               'Schema','Messages',
                               'Table','Topic',
                               'Column',null);
```

This code returns the following results:

```
objname                 value
----------------        ------------------------------------------------------
TopicId                 Surrogate key representing a Topic
Name                    The name of the topic
Description             Description of the purpose and utilization of the topics
RowCreateTime           Time when the row was created
RowLastUpdateTime       Time when the row was last updated
```

There's some pretty cool value in there using extended properties, and not just for documentation. Because the property value is a `sql_variant`, you can put just about anything in there (within a 7,500-character limitation, that is). A possible use could be to store data entry masks and other information that the client could read in once and use to make the client experience richer. In the code download, I have included descriptions for all of the columns in the database.

You aren't limited to tables, column, and schemas either. Constraints, databases, and many other objects in the database can have extended properties. For more information, check the SQL Server 2012 Books Online section "Using Extended Properties on Database Objects."

Viewing the Basic Metadata

In the process of creating a model, knowing where to look in the system metadata for descriptive information about the model is extremely useful. Futzing around in the UI will give you a headache and is certainly not the easiest way to see all of the objects at once, particularly to make sure everything seems to make sense.

There are a plethora of `sys` schema objects. However, they can be a bit messier to use and aren't based on standards, so they're apt to change in future versions of SQL Server, just as these views replaced the system tables from versions of SQL Server before 2005. Of course, with the changes in 2005, it became a lot easier to user the `sys` schema objects (commonly referred to as the system catalog) to get metadata as well. I will stick to the information schema as much as I can because they are based on the SQL standards.

First, let's get a list of the schemas in our database. To view these use the `INFORMATION_SCHEMA.SCHEMATA` view.

```
SELECT SCHEMA_NAME, SCHEMA_OWNER
FROM   INFORMATION_SCHEMA.SCHEMATA
WHERE  SCHEMA_NAME <> SCHEMA_OWNER;
```

Note that I limit the schemas to the ones that don't match their owners. SQL Server automatically creates a schema for every user that gets created.

```
SCHEMA_NAME         SCHEMA_OWNER
------------        ------------
Messages            dbo
Attendees           dbo
```


www.it-ebooks.info

■ **Note** If you are really paying attention, you probably are thinking "didn't he use the `sys.schema` catalog view before?" And yes, that is true. I tend to use the `INFORMATION_SCHEMA` views for reporting on metadata that I want to view, and the catalog views when doing development work, as they can be a bit easier on the eyes since these views include the database name, often as the first column at 128 characters. However, the `INFORMATION_SCHEMA` has a lot of niceties that are useful, and the schema is based on standards so is less likely to change from version to version.

For tables and columns we can use `INFORMATION SCHEMA.COLUMNS`, and with a little massaging, you can see the table, the column name, and the datatype in a format that is easy to use:

```
SELECT table_schema + '.' + TABLE_NAME as TABLE_NAME, COLUMN_NAME,
  --types that have a character or binary lenght
  case when DATA_TYPE IN ('varchar','char','nvarchar','nchar','varbinary')
    then DATA_TYPE + case when character_maximum_length = -1 then '(max)'
      else '(' + CAST(character_maximum_length as
          varchar(4)) + ')' end
    --types with a datetime precision
    when DATA_TYPE IN ('time','datetime2','datetimeoffset')
         then DATA_TYPE + '(' + CAST(DATETIME_PRECISION as varchar(4)) + ')'
    --types with a precision/scale
    when DATA_TYPE IN ('numeric','decimal')
         then DATA_TYPE + '(' + CAST(NUMERIC_PRECISION as varchar(4)) + ',' +
                     CAST(NUMERIC_SCALE as varchar(4)) + ')'
    --timestamp should be reported as rowversion
    when DATA_TYPE = 'timestamp' then 'rowversion'
    --and the rest. Note, float is declared with a bit length, but is
    --represented as either float or real in types
    else DATA_TYPE end as DECLARED_DATA_TYPE,
  COLUMN_DEFAULT
FROM INFORMATION_SCHEMA.COLUMNS
ORDER BY TABLE_SCHEMA, TABLE_NAME,ORDINAL_POSITION
```

which in our database, we have been working on throughout the chapter, returns:

```
TABLE_NAME                    DECLARED_DATA_TYPE     COLUMN_DEFAULT
----------------------------  ---------------------  ----------------
Attendees.AttendeeType        varchar(20)            NULL
Attendees.AttendeeType        varchar(60)            NULL
Attendees.MessagingUser       int                    NULL
Attendees.MessagingUser       varchar(20)            NULL
Attendees.MessagingUser       char(10)               NULL
Attendees.MessagingUser       char(8)                NULL
Attendees.MessagingUser       varchar(50)            NULL
Attendees.MessagingUser       varchar(50)            NULL
Attendees.MessagingUser       varchar(20)            NULL
Attendees.MessagingUser       bit                    ((0))
Attendees.MessagingUser       datetime2(0)           (getdate())
Attendees.MessagingUser       datetime2(0)           (getdate())
```


www.it-ebooks.info

```
Attendees.UserConnection                int                             NULL
Attendees.UserConnection                int                             NULL
Attendees.UserConnection                int                             NULL
Attendees.UserConnection                datetime2(0)                    (getdate())
Attendees.UserConnection                datetime2(0)                    (getdate())
Messages.Message                        int                             NULL
Messages.Message                        datetime2(0)                    NULL
Messages.Message                        int                             NULL
Messages.Message                        int                             NULL
Messages.Message                        nvarchar(200)                   NULL
Messages.Message                        datetime2(0)                    NULL
Messages.Message                        datetime2(0)                    (getdate())
Messages.Message                        datetime2(0)                    (getdate())
Messages.MessageTopic                   int                             NULL
Messages.MessageTopic                   int                             NULL
Messages.MessageTopic                   nvarchar(30)                    NULL
Messages.MessageTopic                   int                             NULL
Messages.MessageTopic                   datetime2(0)                    (getdate())
Messages.MessageTopic                   datetime2(0)                    (getdate())
Messages.Topic                          int                             (NEXT VALUE FOR…
Messages.Topic                          nvarchar(30)                    NULL
Messages.Topic                          varchar(60)                     NULL
Messages.Topic                          datetime2(0)                    (getdate())
Messages.Topic                          datetime2(0)                    (getdate())
```

To see the constraints we have added to these objects (other than the defaults which were included in the previous results, use this code:

```
SELECT TABLE_NAME, CONSTRAINT_NAME, CONSTRAINT_TYPE
FROM   INFORMATION_SCHEMA.table_constraints
WHERE  CONSTRAINT_SCHEMA in ('Attendees','Messages')
ORDER  BY CONSTRAINT_SCHEMA, TABLE_NAME
```

This returns the following results (with the name column truncated for some of the results to fit the data in):

```
TABLE_SCHEMA        TABLE_NAME          CONSTRAINT_NAME                 CONSTRAINT_TYPE
------------        ---------------     ------------------------------  ---------------
Attendees           AttendeeType        PK_Attendees_AttendeeType       PRIMARY KEY
Attendees           MessagingUser       PK_Attendees_MessagingUser      PRIMARY KEY
Attendees           MessagingUser       AK_Attendees_MessagingUser_Us…  UNIQUE
Attendees           MessagingUser       AK_Attendees_MessagingUser_At…  UNIQUE
Attendees           MessagingUser       FK__Attendees_MessagingUser$I…  FOREIGN KEY
Attendees           MessagingUser       CHK__Attendees_MessagingUser_…  CHECK
Attendees           UserConnection      FK__Attendees_MessagingUser$C…  FOREIGN KEY
Attendees           UserConnection      AK_Attendees_UserConnection_U…  UNIQUE
Attendees           UserConnection      PK_Attendees_UserConnection     PRIMARY KEY
Messages            Message             PK_Messages_Message             PRIMARY KEY
Messages            Message             AK_Messages_Message_TimeUserA…  UNIQUE
Messages            Message             FK__Messages_MessagingUser$Se…  FOREIGN KEY
Messages            Message             FK__Messages_MessagingUser$Is…  FOREIGN KEY
```


www.it-ebooks.info

```
Messages            MessageTopic          FK__Messages_Topic$Categorize…      FOREIGN KEY
Messages            MessageTopic          FK__Message$iscCategorizedVia…      FOREIGN KEY
Messages            MessageTopic          AK_Messages_MessageTopic_Topi…      UNIQUE
Messages            MessageTopic          CHK__Messages_MessageTopic_Us…      CHECK
Messages            MessageTopic          PK_Messages_MessageTopic            PRIMARY KEY
Messages            Topic                 PK_Messages_Topic                   PRIMARY KEY
Messages            Topic                 AK_Messages_Topic_Name              UNIQUE
Messages            Topic                 CHK__Messages_Topic_Name_NotE…      CHECK
```

Doing this will minimally help you get an idea of what tables you have create and if you have followed your naming standards. Finally, the following query will give you the list of triggers that have been created.

```
SELECT OBJECT_SCHEMA_NAME(parent_id) + '.' + OBJECT_NAME(parent_id) AS TABLE_NAME,
  name AS TRIGGER_NAME,
  CASE WHEN is_instead_of_trigger = 1 THEN 'INSTEAD OF' ELSE 'AFTER' End
    AS TRIGGER_FIRE_TYPE
FROM sys.triggers
WHERE type_desc = 'SQL_TRIGGER' --not a clr trigger
  AND parent_class = 1 --DML Triggers
ORDER BY TABLE_NAME, TRIGGER_NAME
```

In the text of the chapter, we created three triggers. In the downloads, I have finished the other seven triggers needed to implement the database. The following results include all ten triggers that are included in the downloads:

```
TABLE_NAME                  TRIGGER_NAME                              TRIGGER_FIRE_TYPE
------------------------    ----------------------------------------  -----------------
Attendees.MessagingUser     MessagingUser$InsteadOfInsertTrigger      INSTEAD OF
Attendees.MessagingUser     MessagingUser$UpdateRowControlsTrigger    AFTER
Attendees.UserConnection    UserConnection$InsteadOfInsertTrigger     INSTEAD OF
Attendees.UserConnection    UserConnection$InsteadOfUpdateTrigger     INSTEAD OF
Messages.Message            Message$InsteadOfInsertTrigger            INSTEAD OF
Messages.Message            Message$InsteadOfUpdateTrigger            INSTEAD OF
Messages.MessageTopic       MessageTopic$InsteadOfInsertTrigger       INSTEAD OF
Messages.MessageTopic       MessageTopic$InsteadOfUpdateTrigger       INSTEAD OF
Messages.Topic              Topic$InsteadOfInsertTrigger              INSTEAD OF
Messages.Topic              Topic$InsteadOfUpdateTrigger              INSTEAD OF
```

Finally, if you need to see the check constraints in the database, you can use the following:

```
SELECT TABLE_SCHEMA + '.' + TABLE_NAME AS TABLE_NAME,
       TABLE_CONSTRAINTS.CONSTRAINT_NAME, CHECK_CLAUSE
FROM   INFORMATION_SCHEMA.TABLE_CONSTRAINTS
           JOIN INFORMATION_SCHEMA.CHECK_CONSTRAINTS
               ON TABLE_CONSTRAINTS.CONSTRAINT_SCHEMA =
                       CHECK_CONSTRAINTS.CONSTRAINT_SCHEMA
                   AND TABLE_CONSTRAINTS.CONSTRAINT_NAME = CHECK_CONSTRAINTS.CONSTRAINT_NAME
```


www.it-ebooks.info

This will return

```
TABLE_NAME                CONSTRAINT_NAME                    CHECK_CLAUSE
------------------------  ---------------------------------  -------------------------
Messages.Topic            CHK__Messages_Topic_Name_NotEmpty  (len(rtrim([Name]))>(0))
Messages.MessageTopic     CHK__Messages_MessageTopic_UserDe… (len(rtrim([UserDefinedT…
Attendees.MessagingUser   CHK__Attendees_MessagingUser_User… (len(rtrim([UserHandle])…
Messages.MessageTopic     CHK__Messages_MessageTopic_UserDe… ([UserDefinedTopicName] …
```

This is just a taste of the metadata available, and we will make use of the information schema and other catalog views throughout this book, rather than give you any screen shots of management or developer studio.

■ **Tip** The `INFORMATION_SCHEMA` and catalog views are important resources for the DBA to find out what is in the database. Throughout this book, I will try to give insight into some of them, but there is another book's worth of information out there on the metadata of SQL Server.

Unit Testing Your Structures

Designing tables is a lot of fun, but it is by no means the end of the process. Once you have structures created, you now need to create test scripts to insert good and bad data into your tables to make sure they work. There are some automated tools that are set up to help you with this task (a cursory scan of Sourceforge will show at least five tools that you can use, at a minimum).

Many people have different ideas of testing, particularly trying to treat a database like it is a normal coded object and set up a state for the objects (create or read in some data), try your code, often a stored procedure, then delete the data. This technique works when you are trying to test a single module of code, but it is pretty tedious when you want to test the entire database and you have to load 20 tables to test one.

In this section, I will give you the most simplistic version of testing your database structures. In it, I will use a single script that will run and will basically insert data into your entire database. So far in this chapter, we have created a script to create a completely empty database. This is the database that we will use for our testing. Performance is of no concern, nor is any consideration of concurrency. For this pass of testing, we want to make sure that the database will save and deal with data and will catch data outside of the reasonable norm. I say "reasonable" because unless we have a real reason to do so, we won't be testing minimum and maximum values for a datatype, since we will trust that SQL Server can handle minimum and maximum values. We will also assume that foreign key constraints work to validate insert and updates and will not take time seeing what happens when we violate a constraint with an invalid key value. We will check deletes to make sure that cascading operations work where they need to and not where they do not. We will test any check constraints we have built because these are apt to be an issue, and we will check to make sure that the triggers we have created work as well.

■ **Note** A comprehensive discussion of testing is out of scope for this book as complete testing will require involvement of the entire team. The unit test is generally the first of several testing steps including integration testing, user testing, performance testing, and so on. These will flow out of the application implementation that will be tested simultaneously. During unit testing, the goal will be simply to prove to ourselves that the code we created does the minimum that it is programmed to do.

www.it-ebooks.info

In the text, I will include an abridged version of the test script that you can get with the downloads. We will work with two types of queries. Those we expect to work, and those we expect to fail. For the statements we expect to succeed, we will check the row count after the statement and see if it is what we expect. If an error is raised, that will be self explanatory.

```
<statement to test>
if @@ROWCOUNT <> 1 THROW 50000,'Description of Operation Failed',16;
```

For statements that we expect to fail, we will use a statement that uses a `TRY . . . CATCH` block to capture the error. If no error occurs, the `RAISERROR` statement will force an error to occur. Then in the `CATCH` block, we check to make sure the error message references the constraint we are testing.

```
BEGIN TRY
  <statement to test>
  THROW 50000,'No error raised',16;
END TRY
BEGIN CATCH
  IF ERROR_MESSAGE() NOT LIKE '%<constraint being violated>%'
    THROW 50000,'<Description of Operation> Failed',16;
END CATCH
```

The preceding example is a very minimalist method to test your structures, but even this will take quite a while to build, even for a smallish database. As the number of tables climbs, the complexity rises exponentially because of the likely intertable relationships that have to be violated. The goal is to build a test script that loads up a complete database full of data and tests failures along the way (using our technique to quash errors that are expected) and end up with a full database.

■ **Note** In the download, I have included a script file named `Chapter 6 - Database Create Only.sql` that includes the minimal script to create the database and return the metadata. This will allow you to start with a clean database over and over without working through the entire chapter script.

The first step is to include delete statements to clear out all of the data in the database, except for any data that is part of the base load. The goal here is to make your test script repeatable so you can run your script over and over, particularly if you get an error that you don't expect and you have to go fix your structures.

```
SET NOCOUNT ON;
USE ConferenceMessaging;
GO
DELETE FROM Messages.MessageTopic ;
DELETE FROM Messages.Message;
DELETE FROM Messages.Topic WHERE TopicId <> 0; --Leave the User Defined Topic
DELETE FROM Attendees.UserConnection;
DELETE FROM Attendees.MessagingUser;
```

By deleting the data in the table, you will reset the data, but you won't reset the identity values and the sequence objects. This will help you to make sure that you aren't relying on certain identity values to test with. Next, I will add a legal user to the `MessagingUser` table:

```
INSERT INTO [Attendees].[MessagingUser]
           ([UserHandle],[AccessKeyValue],[AttendeeNumber]
```


www.it-ebooks.info

```
            ,[FirstName],[LastName],[AttendeeType]
            ,[DisabledFlag])
VALUES ('FredF','0000000000','00000000','Fred','Flintstone','Regular',0);

if @@ROWCOUNT <> 1 THROW 50000,'Attendees.MessagingUser Single Row Failed',16;
```

Next, we will test entering data that fails one of the check constraints. In the next statement, I will enter data with a user handle that is too small:

```
BEGIN TRY --Check UserHandle Check Constraint
  INSERT INTO [Attendees].[MessagingUser]
    ([UserHandle],[AccessKeyValue],[AttendeeNumber]
    ,[FirstName],[LastName],[AttendeeType]
    ,[DisabledFlag])
  VALUES ('Wil','0000000000','00000001','Wilma','Flintstone','Regular',0);
  THROW 50000,'No error raised',16,1;
END TRY
BEGIN CATCH
  if ERROR_MESSAGE() not like
    '%CHK__Attendees_MessagingUser_UserHandle_LenthAndStart%'
  THROW 50000,'Check Messages.Topic.Name didn''t work',16;
END CATCH
```

When you execute this statement, you won't get an error if the constraint you expect to fail is mentioned in the error message (and it will be if you have built the same database I have). Then, I will enter another row that fails the check constraint due to use of a non-alphanumeric character in the handle:

```
BEGIN TRY --Check UserHandle Check Constraint
  INSERT INTO [Attendees].[MessagingUser]
    ([UserHandle],[AccessKeyValue],[AttendeeNumber]
    ,[FirstName],[LastName],[AttendeeType]
    ,[DisabledFlag])
  VALUES ('Wilma@','0000000000','00000001','Wilma','Flintstone','Regular',0);
  THROW 50000,'No error raised',16,1;
END TRY
BEGIN CATCH
    if ERROR_MESSAGE() not like
    '%CHK__Attendees_MessagingUser_UserHandle_LenthAndStart%'
  THROW 50000,'Check Messages.Topic.Name didn''t work',16;
END CATCH
GO
```

■ **Note** This method of unit testing can be a bit confusing when casually checking things. In the previous block of code, the statement fails, but no error is returned. The goal is that you can run your test script over and over and getting no output other then seeing rows in your tables. However, in practice, it is a lot cleaner to see only problematic output. If you would prefer, add more output to your test script as best suits your desire.

---

www.it-ebooks.info

Skipping some of the simpler test items we now arrive at a test of the unique constraint we set up based on the RoundedMessageTime that rounds the MessageTime to the hour. (Some of the data to support these tests are included in the sample code. You can search for the comments in the example code to find our place in the download). To test this, I will enter a row into the table and then immediately enter another at exactly the same. If you happen to run this on a slow machine right at the turn of the hour, although it is extremely unlikely,the two statements execute in the same second (probably even the same millisecond).

```
INSERT INTO [Messages].[Message]
            ([MessagingUserId]
            ,[SentToMessagingUserId]
            ,[Text]
            ,[MessageTime])
  VALUES
  ((SELECT MessagingUserId FROM Attendees.MessagingUser WHERE UserHandle = 'FredF')
  ,(SELECT MessagingUserId FROM Attendees.MessagingUser WHERE UserHandle = 'WilmaF')
  ,'It looks like I will be late tonight'
  ,GETDATE());
IF @@ROWCOUNT <> 1 THROW 50000,'Messages.Messages Single Insert Failed',16;
GO
```

Then, this statement will cause an error that should be caught in the CATCH block:

```
BEGIN TRY --Unique Message Error…
  INSERT INTO [Messages].[Message]
                       ([MessagingUserId]
                       ,[SentToMessagingUserId]
                       ,[Text]
           ,[MessageTime])
    VALUES
    ((SELECT MessagingUserId FROM Attendees.MessagingUser  WHERE UserHandle = 'FredF')
    ,(SELECT MessagingUserId FROM Attendees.MessagingUser  WHERE UserHandle = 'WilmaF') --
    ,'It looks like I will be late tonight'
    ,GETDATE())
  THROW 50000,'No error raised',16;
END TRY
BEGIN CATCH
    if ERROR_MESSAGE() NOT LIKE '%AK_Messages_Message_TimeUserAndText%'
           THROW 50000,'Unique Message Error didn''t work (check times)',16;
END CATCH
GO
```

If the error occurs, it is trapped and we know the constraint is working. If no error occurs, then the no error THROW will. Finally, I will show in the text the most complicated error checking block we have to deal with. This is the message, and the message Topic. In the download, I insert the two successful cases, first for a specific topic, then with a user defined topic. In the next block, I will show the failure case.

```
--Do this in a more natural way. Usually the client would pass in these values
DECLARE @messagingUserId int, @text nvarchar(200),
        @messageTime datetime2, @RoundedMessageTime datetime2(0);

SELECT @messagingUserId = (SELECT MessagingUserId FROM Attendees.MessagingUser
                           WHERE UserHandle = 'FredF'),
       @text = 'Oops Why Did I say That?', @messageTime = SYSDATETIME();
```


www.it-ebooks.info

```
--uses the same algorithm as the check constraint to calculate part of the key
SELECT @RoundedMessageTime = (
DATEADD(HOUR,DATEPART(HOUR,@MessageTime),CONVERT(datetime2(0),CONVERT(date,@MessageTime))));

BEGIN TRY
  BEGIN TRANSACTION
      --first create a new message
      INSERT INTO [Messages].[Message]
      ([MessagingUserId],[SentToMessagingUserId]
      ,[Text] ,[MessageTime])
  VALUES
  (@messagingUserId,NULL,@text, @messageTime);
  --then insert the topic,but this will fail because General topic is not
  --compatible with a UserDefinedTopicName value
  INSERT INTO Messages.MessageTopic(MessageId, TopicId, UserDefinedTopicName)
    VALUES(
    (SELECT MessageId
    FROM    Messages.Message
    WHERE   MessagingUserId = @messagingUserId
    AND Text = @text
    AND RoundedMessageTime = @RoundedMessageTime),
        (SELECT TopicId
        FROM Messages.Topic
        WHERE Name = 'General'),'Stupid Stuff');

  COMMIT TRANSACTION
END TRY
BEGIN CATCH
  if ERROR_MESSAGE() not like
  '%CHK__Messages_MessageTopic_UserDefinedTopicName_NullUnlessUserDefined%'
     THROW 50000,'User Defined Message Check Failed',16;
END CATCH
```

The test script provided with the download is just a very basic example of a test script, and it will take a while to get a good unit test script created. It took me more than a several hours to create this one for this simple six-table database. Inserting data in a script like this, you will have to do some ugly coding to make it work. In reality, I usually start with a simple script that creates data only without testing the check constraints, unique constraints, and so on, because it is rare that you have time to do the complete test script before turning the database over to developers.

Once the process of building your unit tests is completed, you will find that it will have helped you find issues with your design and any problems with constraints. In many cases, you may not want to put certain constraints on the development server immediately and work with developers to know when they are ready. As a DB developer, and a lapsed UI developer, I personally liked it when the database prevented me from breaking a fundamental rule, so your mileage may vary as to what works best. I will say this, as I created this script, I discovered a few semi-significant issues with the demo design I created for this chapter that wouldn't have likely been noticed except by you as you, the intrepid reader, work through the design.

Best Practices

The following are a set of some of the most important best practices when implementing your database structures. Pay particular attention to the advice about UNIQUE constraints. Just having a surrogate key on a table is one of the worst mistakes made when implementing a database.

www.it-ebooks.info

- *Invest in database generation tools*: Do this after you know what the tool should be doing (not before). Implementing tables, columns, relationships, and so on is a tedious and painful task when done by hand. There are many great tools that double as logical data modeling tools and also generate these objects, as well as sometimes the objects and code to be covered in the upcoming three chapters.
- *Maintain normalization*: As much as possible, try to maintain the normalizations that were designed in Chapter 5. It will help keep the data better protected and will be more resilient to change.
- *Develop a real strategy for naming objects*: Keep the basics in mind:
  - Give all objects reasonably user-friendly names. Make sure that it's obvious—at least to support personnel—what the purpose of every object, column, and so on is without digging into documentation, if at all possible.
  - Have either all plural or all singular names for tables. Consistency is the key.
  - Have all singular names for columns.
  - I use singular names for tables or columns.
- *Develop template domains*: Reuse in every case where a similar datatype is needed. This cuts down on time spent implementing and makes users of the data happy, because every time they see a column called `Description`, it's likely that it will have the same characteristics of other like columns.
- *Carefully choose the datatype and nullability for each column*: These are the first level of data protection to keep your data clean and pure. Also, improper datatypes can cause precision difficulties with numbers and even performance issues.
- *Make certain that every table has at least one UNIQUE constraint that doesn't include an artificial value*: It's a good idea to consider using an `IDENTITY` column as the primary key. However, if that is the only `UNIQUE` constraint on the table, then there can (and usually will) be duplication in the **real** columns of the table—a bad idea.
- *Implement foreign keys using foreign key constraints*: They're fast, and no matter what kind of gaffes a client makes, the relationship between tables cannot be gotten wrong if a foreign key constraint is in place.
- *Document and script everything*: Using extended properties to document your objects can be extremely valuable. Most of all, when you create objects in the database, keep scripts of the T-SQL code for later use when moving to the QA and production environments. A further step of keeping your scripts in a source control repository is a definite good next step as well so you can see where you are, where you are going, and where you have been in one neat location.
- *Develop a test script*: Test your structures as much as possible. Testing is often the forgotten step in database design, but good testing is essential to know your design works.

Summary

This has been a long chapter covering a large amount of ground. Understanding how to build tables, and how they're implemented, is the backbone of every database designer's knowledge.

www.it-ebooks.info

After getting satisfied that a model was ready to implement, I took a deep look at SQL Server tables, walking through the process of creating the database using the `CREATE TABLE` and `ALTER TABLE` syntax for adding constraints and modifying columns, and even created triggers to manage automatically maintained columns. General guidelines were given for naming tables, columns, indexes, and foreign key constraints. The key to good naming is consistency, and if the naming standards I suggested here seem too ugly, messy, or just plain weird to you, choose your own. Consistency is the most important thing for any naming convention.

The two most important sections of this chapter were on choosing datatypes and implementing unique keys. I completed this chapter by discussing the process of choosing primary keys and at least one natural key per table. Of course, the section on testing is pretty important too as good testing is the key to finding those obvious errors that some developers will eagerly point out to anyone will listen to about how much slower the process is with constraints, triggers, and such.

In the next chapter, I'll show how to finish the task of implementing the base OLTP system by implementing the rest of the business rules required to keep the data in your database as clean as possible.

www.it-ebooks.info

CHAPTER 7

Data Protection with Check Constraints and Triggers

You can't, in sound morals, condemn a man for taking care of his own integrity. It is his clear duty.

—Joseph Conrad

One of the weirdest things I see in database implementations is that people spend tremendous amounts of time designing the correct database storage (or, at least, what seems like tremendous amounts of time to them) and then just leave the data unprotected with tables being more or less treated like buckets that will accept anything, opting to let code outside of the database layer to do all of the data protection. Honestly, I do understand the allure, in that the more constraints you apply, the harder development is in the early stages of the project, and the programmers honestly do believe that they will catch everything. The problem is, there is rarely a way to be 100% sure that all code written will always enforce every rule.

The second argument against using automatically enforced data protection is that programmers want complete control over the errors they will get back and over what events may occur that can change data. Later in this chapter, I will suggest methods that will let data formatting or even cascading insert operations occur to make sure that certain conditions in the data are met, making coding a lot easier. While the data being manipulated "magically" can be confusing initially, you have to think of the data layer as part of the application, not as a bucket with no limitations. Keeping the data from becoming an untrustworthy calamity of random bits is in everyone's best interest.

Perhaps, in an ideal world, you could control all data input carefully, but in reality, the database is designed and then turned over to the programmers and users to "do their thing." Those pesky users immediately exploit any weakness in your design to meet the requirements that they "*thought* they gave you in the *first* place." No matter how many times I've forgotten to apply a `UNIQUE` constraint in a place where one was natural to be (yeah, I am preaching to myself along with the choir in this book sometimes), it's amazing to me how quickly the data duplications start to occur. Ultimately, user perception is governed by the reliability and integrity of the data users retrieve from your database. If they detect data anomalies in their data sets (usually in skewed report values), their faith in the whole application plummets faster than a skydiving elephant who packed lunch instead of a parachute. After all, your future reputation is based somewhat on the perceptions of those who use the data on a daily basis.

One of the things I hope you will feel as you read this chapter (and keep the earlier ones in mind) is that, if at all possible, the data storage layer should own protection of the fundamental data integrity. Not that the other code shouldn't play a part: I don't want to have to wait for the database layer to tell me that a required value is missing, but at the same time, I don't want a back-end loading process to have to use application code to validate

www.it-ebooks.info

that the data is fundamentally correct either. If the column allows `NULL`s, then I should be able to assume that a `NULL` value is at least in some context allowable. If the column is a `nvarchar(20)` column with no other data checking, I should be able to put every Unicode character in the column, and up to 20 concatenated values at that. The primary point of data protection is that the application layers ought do a good job of making it easy for the user, but the data layer can realistically be made nearly 100 percent trustworthy, whereas the application layers cannot. At a basic level, you expect keys to be validated, data to be reasonably formatted and fall within acceptable ranges, and required values to always exist, just to name a few. When those criteria can be assured, the rest won't be so difficult, since the application layers can trust that the data they fetch from the database meets them, rather than having to revalidate.

The reason I like to have the data validation and protection logic as close as possible to the data it guards is that it has the advantage that you *have* to write this logic only once. It's all stored in the same place, and it takes forethought to bypass. At the same time, I believe you should implement all data protection rules in the client, including the ones you have put at the database-engine level. This is mostly for software usability sake, as no user wants to have to wait for the round-trip to the server to find out that a column value is required when the UI could have simply indicated this to them, either with a simple message or even with visual cues. You build these simple validations into the client, so users get immediate feedback. Putting code in multiple locations like this bothers a lot of people because they think it's

- Bad for performance
- More work

As C.S. Lewis had one of his evil characters note, "By mixing a little truth with it they had made their lie far stronger." The fact of the matter is that these are, in fact, true statements, but in the end, it is a matter of degrees. Putting code in several places is a bit worse on performance, usually in a minor way, but done right, it will help, rather than hinder, the overall performance of the system. Is it more work? Well, initially it is for sure. I certainly can't try to make it seem like it's less work to do something in multiple places, but I can say that it is completely worth doing. In a good user interface, you will likely code even simple rules in multiple places, such as having the color of a column indicate that a value is required and having a check in the submit button that looks for a reasonable value instantly before trying to save the value where it is again checked by the business rule or object layer.

The real problem we must solve is that data can come from multiple locations:

- Users using custom front-end tools
- Users using generic data manipulation tools, such as Microsoft Access
- Routines that import data from external sources
- Raw queries executed by data administrators to fix problems caused by user error

Each of these poses different issues for your integrity scheme. What's most important is that each of these scenarios (with the possible exception of the second) forms part of nearly every database system developed. To best handle each scenario, the data must be safeguarded, using mechanisms that work without the responsibility of the user.

If you decide to implement your data logic in a different tier other than directly in the database, you have to make sure that you implement it—and far more importantly, implement it *correctly*—in every single one of those clients. If you update the logic, you have to update it in multiple locations anyhow. If a client is "retired" and a new one introduced, the logic must be replicated in that new client. You're much more susceptible to coding errors if you have to write the code in more than one place. Having your data protected in a single location helps prevent programmers from forgetting to enforce a rule in one situation, even if they remember *everywhere* else.

Because of concurrency, every statement is apt to fail due to a deadlock, or a timeout, or the data validated in the UI no longer being in the same state as it was even milliseconds ago. In Chapter 11, we will cover concurrency, but suffice it to say that errors arising from issues in concurrency are often exceedingly random in

www.it-ebooks.info

appearance and must be treated as occurring at any time. And concurrency is the final nail in the coffin of using a client tier only for integrity checking. Unless you elaborately lock all users out of the database objects you are using, the state could change and a database error could occur. Are the errors annoying? Yes, they are, but they are the last line of defense between having excellent data integrity and something quite the opposite.

In this chapter, I will look at the two basic building blocks of enforcing data integrity in SQL Server, first using declarative objects: check constraints, which allow you to define predicates on new rows in a table, and triggers, which are stored procedure style objects that can fire after a table's contents have changed.

Check Constraints

Check constraints are part of a class of the declarative constraints that are a part of the base implementation of a table. Basically, constraints are SQL Server devices that are used to enforce data integrity automatically on a single column or row. You should use constraints as extensively as possible to protect your data, because they're simple and, for the most part, have minimal overhead.

One of the greatest aspects of all of SQL Server's constraints (other than defaults) is that the query optimizer can use them to optimize queries, because the constraints tell the optimizer about some additional quality aspect of the data. For example, say you place a constraint on a column that requires that all values for that column must fall between 5 and 10. If a query is executed that asks for all rows with a value greater than 100 for that column, the optimizer will know without even looking at the data that no rows meet the criteria.

SQL Server has five kinds of declarative constraints:

- `NULL`: Determines if a column will accept `NULL` for its value. Though `NULL` constraints aren't technically constraints, they behave like them.
- `PRIMARY KEY` and `UNIQUE` constraints: Used to make sure your rows contain only unique combinations of values over a given set of key columns.
- `FOREIGN KEY`: Used to make sure that any migrated keys have only valid values that match the key columns they reference.
- `DEFAULT`: Used to set an acceptable default value for a column when the user doesn't provide one. (Some people don't count defaults as constraints, because they don't constrain updates.)
- `CHECK`: Used to limit the values that can be entered into a single column or an entire row.

We have introduced `NULL`, `PRIMARY KEY`, `UNIQUE`, and `DEFAULT` constraints in enough detail in Chapter 6; they are pretty straightforward without a lot of variation in the ways you will use them. In this section, I will focus the examples on the various ways to use check constraints to implement data protection patterns for your columns/rows. You use `CHECK` constraints to disallow improper data from being entered into columns of a table. `CHECK` constraints are executed after `DEFAULT` constraints (so you cannot specify a default value that would contradict a `CHECK` constraint) and `INSTEAD OF` triggers (covered later in this chapter) but before AFTER triggers. `CHECK` constraints cannot affect the values being inserted or deleted but are used to verify the validity of the supplied values.

The biggest complaint that is often lodged against constraints is about the horrible error messages you will get back. It is one of my biggest complaints as well, and there is very little you can do about it, although I will posit a solution to the problem later in this chapter. It will behoove you to understand one important thing: all statements should have error handling as if the database might give you back an error—because it might.

There are two flavors of `CHECK` constraint: column and table. Column constraints reference a single column and are used when the individual column is referenced in a modification. `CHECK` constraints are considered table constraints when more than one column is referenced in the criteria. Fortunately, you don't have to worry about declaring a constraint as either a column constraint or a table constraint. When SQL Server compiles the constraint, it verifies whether it needs to check more than one column and sets the proper internal values.

www.it-ebooks.info

We'll be looking at building CHECK constraints using two methods:

- Simple expressions
- Complex expressions using user-defined functions

The two methods are similar, but you can build more complex constraints using functions, though the code in a function can be more complex and difficult to manage. In this section, we'll take a look at some examples of constraints built using each of these methods; then we'll take a look at a scheme for dealing with errors from constraints. First, though, let's set up a simple schema that will form the basis of the examples in this section.

The examples in this section on creating CHECK constraints use the sample tables shown in Figure 7-1.

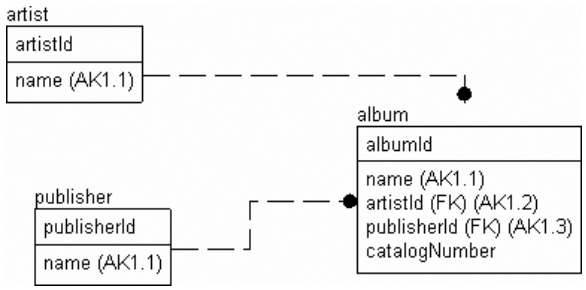


Figure 7-1. *The example schema*

To create and populate the tables, execute the following code (in the downloads, I include a simple create database for a database named Chapter7 and will put all objects in that database):

```
CREATE SCHEMA Music;
GO

CREATE TABLE Music.Artist
(
  ArtistId int NOT NULL,
  Name varchar(60) NOT NULL,
  CONSTRAINT PKMusic_Artist PRIMARY KEY CLUSTERED (ArtistId),
  CONSTRAINT PKMusic_Artist_Name UNIQUE NONCLUSTERED (Name)
);

CREATE TABLE Music.Publisher
(
  PublisherId    int NOT NULL, PRIMARY KEY
  Name           varchar(20) NOT NULL,
  CatalogNumberMask varchar(100) NOT NULL
  CONSTRAINT DfltMusic_Publisher_CatalogNumberMask DEFAULT ('%'),
  CONSTRAINT AKMusic_Publisher_Name UNIQUE NONCLUSTERED (Name),
);

CREATE TABLE Music.Album
(
  AlbumId int NOT NULL,
  Name varchar(60) NOT NULL,
  ArtistId int NOT NULL,
```


www.it-ebooks.info

```
  CatalogNumber varchar(20) NOT NULL,
  PublisherId int NOT NULL --not requiring this information

  CONSTRAINT PKMusic_Album PRIMARY KEY CLUSTERED(AlbumId),
  CONSTRAINT AKMusic_Album_Name UNIQUE NONCLUSTERED (Name),
  CONSTRAINT FKMusic_Artist$records$Music_Album
      FOREIGN KEY (ArtistId) REFERENCES Music.Artist(ArtistId),
  CONSTRAINT FKMusic_Publisher$published$Music_Album
      FOREIGN KEY (PublisherId) REFERENCES Music.Publisher(PublisherId)
);
```

Then seed the tables with the following data:

```
INSERT INTO Music.Publisher (PublisherId, Name, CatalogNumberMask)
VALUES (1,'Capitol',
       '[0-9][0-9][0-9]-[0-9][0-9][0-9a-z][0-9a-z][0-9a-z]-[0-9][0-9]'),
       (2,'MCA', '[a-z][a-z][0-9][0-9][0-9][0-9][0-9]');

INSERT INTO Music.Artist(ArtistId, Name)
VALUES (1, 'The Beatles'),(2, 'The Who');

INSERT INTO Music.Album (AlbumId, Name, ArtistId, PublisherId, CatalogNumber)
VALUES (1, 'The White Album',1,1,'433-43ASD-33'),
       (2, 'Revolver',1,1,'111-11111-11'),
       (3, 'Quadrophenia',2,2,'CD12345');
```

A likely problem with this design is that it isn't normalized well enough for a realistic solution. Publishers usually have a mask that's valid at a given point in time, but everything changes. If the publishers lengthen the size of their catalog numbers or change to a new format, what happens to the older data? For a functioning system, it would be valuable to have a release-date column and catalog number mask that was valid for a given range of dates. Of course, if you implemented the table as presented, the enterprising user, to get around the improper design, would create publisher rows such as `'MCA 1989-1990'`, `'MCA 1991-1994'`, and so on and mess up the data for future reporting needs, because then, you'd have work to do to correlate values from the MCA company (and your table would be not even technically in First Normal Form!).

As a first example of a check constraint, consider if you had a business rule that no artist with a name that contains the word `'Pet'` *followed by the word* `'Shop'` is allowed, you could code the following as follows (note, all examples assume a case-insensitive collation, which is almost certainly the norm):

```
ALTER TABLE Music.Artist WITH CHECK
  ADD CONSTRAINT chkMusic_Artist$Name$NoPetShopNames
      CHECK (Name not like '%Pet%Shop%');
```

Then, test by trying to insert a new row with an offending value:

```
INSERT INTO Music.Artist(ArtistId, Name)
VALUES (3, 'Pet Shop Boys');
```

This returns the following result

```
Msg 547, Level 16, State 0, Line 1

The INSERT statement conflicted with the CHECK constraint "chkMusic_Artist$Name$NoPetShopNames".
The conflict occurred in database "Chapter7", table "Music.Artist", column 'Name'.
```


www.it-ebooks.info

thereby keeping my music collection database safe from at least one band from the '80s.

When you create a CHECK constraint, the WITH NOCHECK setting (the default is WITH CHECK) gives you the opportunity to add the constraint without checking the existing data in the table.

Let's add a row for another musician who I don't necessarily want in my table:

```
INSERT INTO Music.Artist(ArtistId, Name)
VALUES (3, 'Madonna');
```

Later in the process, it is desired that no artists with the word "Madonna" will be added to the database, but if you attempt to add a check constraint

```
ALTER TABLE Music.Artist WITH CHECK
  ADD CONSTRAINT chkMusic_Artist$Name$noMadonnaNames
    CHECK (Name NOT LIKE '%Madonna%');
```

rather than the happy "Command(s) completed successfully." message you so desire to see, you see the following:

```
Msg 547, Level 16, State 0, Line 1

The ALTER TABLE statement conflicted with the CHECK constraint "chkMusic_
Artist$Name$noMadonnaNames". The conflict occurred in database "Chapter7",
table "Music.Artist", column 'Name'.
```

In order to allow the constraint to be added, you might specify the constraint using WITH NOCHECK rather than WITH CHECK because you now want to allow this new constraint, but there's data in the table that conflicts with the constraint, and it is deemed too costly to fix or clean up the existing data.

```
ALTER TABLE Music.Artist WITH NOCHECK
  ADD CONSTRAINT chkMusic_Artist$Name$noMadonnaNames
    CHECK (Name NOT LIKE '%Madonna%');
```

The statement is executed to add the check constraint to the table definition, though using NOCHECK means that the bad value does not affect the creation of the constraint. This is OK in some cases but can be very confusing because anytime a modification statement references the column, the CHECK constraint is fired. The next time you try to set the value of the table to the same bad value, an error occurs. In the following statement, I simply set every row of the table to the same name it has stored in it:

```
UPDATE Music.Artist
SET Name = Name;
```

This gives you the following error message:

```
Msg 547, Level 16, State 0, Line 1

The UPDATE statement conflicted with the CHECK constraint "chkMusic_Artist$Name$noMadonnaNames".
The conflict occurred in database "Chapter7", table "Music.Artist", column 'Name'.
```

"What?" most users will exclaim. If the value was in the table, shouldn't it already be good? The user is correct. This kind of thing will confuse the heck out of everyone and cost you greatly in support, unless the data in question is *never* used. But if it's never used, just delete it, or include a time range for the values. CHECK Name NOT LIKE %Madonna% OR rowCreateDate < '20111131' could be a reasonable compromise. Using NOCHECK and leaving the values unchecked is almost worse than leaving the constraint off in many ways.

www.it-ebooks.info

■ **Tip** If a data value could be right or wrong, based on external criteria, it is best not to be overzealous in your enforcement. The fact is, unless you can be 100 percent sure, when you use the data later, you will still need to make sure that the data is correct before usage.

One of the things that makes constraints excellent beyond the obvious data integrity reasons is that if the constraint is built using `WITH CHECK`, the optimizer can make use of this fact when building plans if the constraint didn't use any functions and just used simple comparisons such as less than, greater than, and so on. For example, imagine you have a constraint that says that a value must be less than or equal to 10. If, in a query, you look for all values of 11 and greater, the optimizer can use this fact and immediately return zero rows, rather than having to scan the table to see whether any value matches.

If a constraint is built with `WITH CHECK`, it's considered trusted, because the optimizer can trust that all values conform to the `CHECK` constraint. You can determine whether a constraint is trusted by using the `sys.check_constraints` catalog object:

```
SELECT definition, is_not_trusted
FROM   sys.check_constraints
WHERE  object_schema_name(object_id) = 'Music'
  AND name = 'chkMusic_Artist$Name$noMadonnaNames';
```

This returns the following results (with some minor formatting, of course):

```
definition                      is_not_trusted
------------------------------  -------------
(NOT [Name] LIKE '%Madonna%')   1
```

Make sure, if at all possible, that `is_not_Trusted = 0` for all rows so that the system trusts all your `CHECK` constraints and the optimizer can use the information when building plans.

■ **Caution** Creating CHECK constraints using the `CHECK` option (instead of `NOCHECK`) on a tremendously large table can take a very long time to apply, so often, you'll feel like you need to cut corners to get it done fast. The problem is that the shortcut on design or implementation often costs far more in later maintenance costs or, even worse, in the user experience. If at all possible, it's best to try to get everything set up properly, so there is no confusion.

To make the constraint trusted, you will need to clean up the data and use `ALTER TABLE <tableName> WITH CHECK CHECK CONSTRAINT constraintName` to have SQL Server check the constraint and set it to trusted. Of course, this method suffers from the same issues as creating the constraint with `NOCHECK` in the first place (mostly, it can take forever!). But without checking the data, the constraint will not be trusted, not to mention that forgetting to reenable the constraint is too easy. For our constraint, we can try to check the values:

```
ALTER TABLE Music.Artist WITH CHECK CHECK CONSTRAINT chkMusic_Artist$Name$noMadonnaNames;
```


www.it-ebooks.info

And it will return the following error (as it did when we tried to create it the first time):

```
Msg 547, Level 16, State 0, Line 1

The ALTER TABLE statement conflicted with the CHECK constraint "chkMusic_
Artist$Name$noMadonnaNames". The conflict occurred in database "Chapter7", table "Music.
Artist", column 'Name'.
```

But, if we delete the row with the name Madonna

```
DELETE FROM Music.Artist
WHERE Name = 'Madonna';
```

and try again, the `ALTER TABLE` statement will be execute without error, and the constraint will be trusted (and all will be well with the world!). One last thing you can do is to disable a constraint, using `NOCHECK`:

```
ALTER TABLE Music.Artist NOCHECK CONSTRAINT chkMusic_Artist$Name$noMadonnaNames;
```

Now, you can see that the constraint is disabled by adding an additional object property:

```
SELECT definition, is_not_trusted, is_disabled
FROM    sys.check_constraints
WHERE   OBJECT_SCHEMA_NAME(object_id) = 'Music'
  AND name = 'chkMusic_Artist$Name$noMadonnaNames';
```

which will return

```
definition                      is_not_trusted      is_disabled
-----------------------------   --------------      -----------
(NOT [Name] LIKE '%Madonna%')   1                   1
```

Then, rerun the statement to enable the statement before we continue:

```
ALTER TABLE Music.Artist WITH CHECK CHECK CONSTRAINT chkMusic_Artist$Name$noMadonnaNames
```

After that, checking the output of the `sys.check_constraints` query, you will see that it has been enabled.

CHECK Constraints Based on Simple Expressions

By far, most `CHECK` constraints are simple expressions that just test some characteristic of a value in a column or columns. These constraints often don't reference any data other than the single column but can reference any of the columns in a single row.

As a few examples, consider the following:

- *Empty strings*: Prevent users from inserting one or more space characters to avoid any real input into a column—`CHECK (LEN(ColumnName) > 0)`. *This constraint is on 90 percent of the* `varchar` *and* `char` *columns in databases I design.*
- *Date range checks*: Make sure a reasonable date is entered, for example:
    - The date a rental is required to be returned should be greater than one day after the `RentalDate` (assume the two columns are implemented with the date datatype): `CHECK (ReturnDate > dateadd(day,1,RentalDate))`.

www.it-ebooks.info

    - Date of some event that's supposed to have occurred already in the past: `CHECK (EventDate <= GETDATE())`.

- *Value reasonableness*: Make sure some value, typically a number of some sort, is reasonable for the situation. Reasonable, of course, does not imply that the value is necessarily correct for the given situation, which is usually the domain of the middle tier of objects—just that it is within a reasonable domain of values. For example:

    - Values must be a nonnegative integer. This is common, because there are often columns where negative values don't make sense (hours worked, miles driven, and so on): `CHECK (MilesDriven >= 0)`.

    - Royalty rate for an author that's less than or equal to 30 percent. If this rate ever could be greater, it isn't a `CHECK` constraint. So if 15 percent is the typical rate, the UI might warn that it isn't normal, but if 30 percent is the absolute ceiling, it would be a good `CHECK` constraint: `CHECK (RoyaltyRate <= .3)`.

`CHECK` constraints of this variety are always a good idea when you have situations where there are data conditions that *must* always be true. Another way to put this is that the very definition of the data is being constrained, not just a convention that could change fairly often or even be situationally different. These `CHECK` constraints are generally extremely fast and won't negatively affect performance except in extreme situations. As an example, I'll just show the code for the first, empty string check, because simple `CHECK` constraints are easy to code once you have the syntax. A common `CHECK` constraint that I add to string type columns (`varchar`, `char`) prevents blank data from being entered. This is because, most of the time, if a value is required, it isn't desired that the value for a column be blank, unless having no value for the column makes sense(as opposed to having a `NULL` value, meaning that the value is not currently known).

For example, in the `Album` table, the `Name` column doesn't allow `NULL`s. The user has to enter something, but what about when the enterprising user realizes that `''` is not the same as `NULL`? What will be the response to an empty string? Ideally, of course, the UI wouldn't allow such nonsense for a column that had been specified as being required, but the user just hits the space bar, but to make sure, we will want to code a constraint to avoid it.

To avoid letting a user get away with a blank row, you can add the following constraint to prevent this from ever happening again (after deleting the two blank rows). It works by using the `LEN` function that does a trim by default, eliminating any space characters, and checking the length:

```
ALTER TABLE Music.Album WITH CHECK
  ADD CONSTRAINT chkMusicAlbum$Name$noEmptyString
    CHECK (LEN(Name) > 0); --note,len does a trim by default, so any string
                           --of all space characters will return 0
```

Testing this with data that will clash with the new constraint

```
INSERT INTO Music.Album ( AlbumId, Name, ArtistId, PublisherId, CatalogNumber )
VALUES ( 4, '', 1, 1,'dummy value' );
```

you get the following error message

```
Msg 547, Level 16, State 0, Line 1

The INSERT statement conflicted with the CHECK constraint "chkMusicAlbum$Name$noEmptyString".
The conflict occurred in database "Chapter7", table "Music.Album", column 'Name'.
```

All too often, nonsensical data is entered just to get around your warning, but that is more of a UI or managerial oversight problem than a database design concern, because the check to see whether `'ASDFASDF'` is a reasonable name value is definitely not of the definite true/false variety. (Have you seen what some people name

www.it-ebooks.info

their kids?) What's generally the case is that the user interface will then prevent such data from being created via the UI, but the CHECK constraint is there to prevent other processes from putting in completely invalid data as well.

These check constraints are very useful when you are loading data into the table from an outside source. Often, when data is imported from a file, like from the Import Wizard, blank data will be propagated as blank values, and the programmers involved might not think to deal with this condition. The check constraints make sure that the data is put in correctly. And as long as you are certain to go back and recheck the trusted status and values, their existence helps to remind you even if they are ignored, like using SSIS's bulk loading features. In Figure 7-2, you will see that you can choose to (or choose not to) check constraints on the OLEDB destination output. In this case, it may either disable the constraint, or set to not trusted to speed loading, but it will limit the data integrity and optimizer utilization of the constraint until you reset it to trusted as was demonstrated in the previous section.

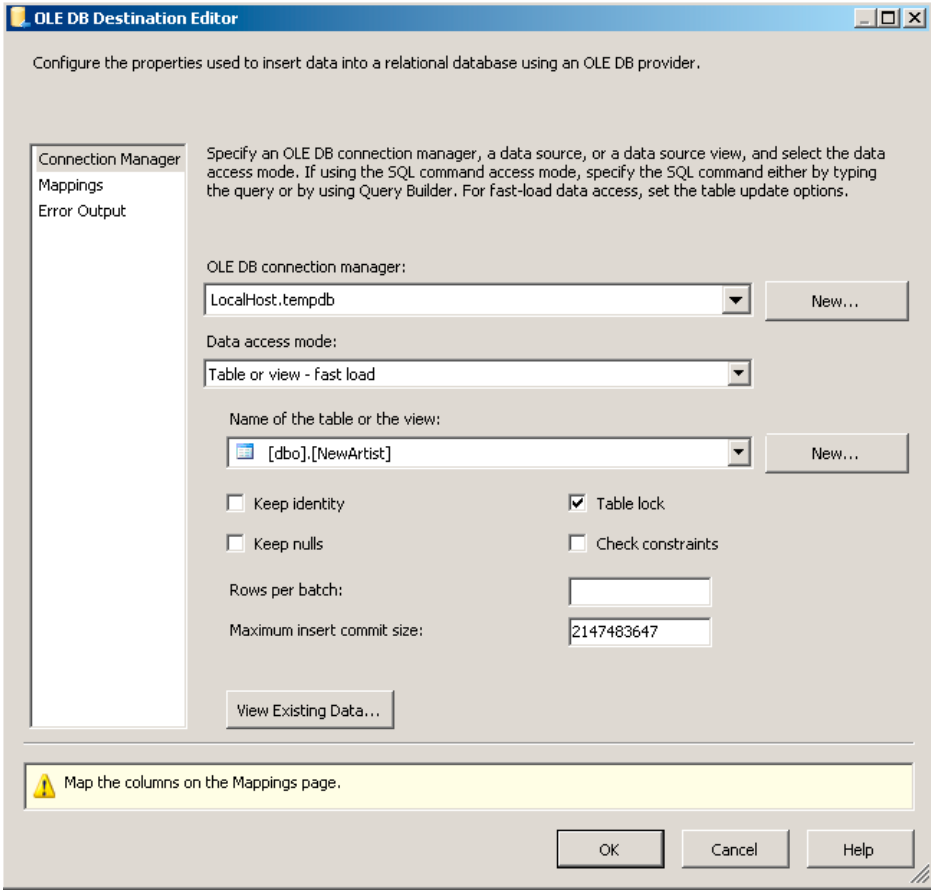


Figure 7-2. *Example SSIS OLEDB Output with Check constraints deselected*

CHECK Constraints Using Functions

Sometimes, you really need to implement a complex data check where a simple Boolean expression using the columns in the table and base T-SQL scalar functions just won't do. In standard SQL, you can use a subquery in

www.it-ebooks.info

your constraints. However, in SQL Server, subqueries are not allowed, but you can use a scalar T-SQL function, even if it accesses another table.

In general, using functions is a fairly atypical solution to ensuring data integrity, but it can be far more powerful and, in many ways, quite useful when you need to build slightly complex data integrity protection. For the most part, CHECK constraints usually consist of the simple task of checking a stable format or value of a single column, and for these tasks, a standard CHECK constraint using the simple <BooleanExpression> is perfectly adequate.

However, a CHECK constraint need not be so simple. A UDF can be complex and might touch several tables in the instance. Here are some examples:

- *Complex scalar validations (often using CLR functions):* For example, in a situation where a regular expression would be easier to use than a LIKE comparison.
- *Validations that access other tables*: For example, to check a domain that is based on values in several tables, rather than a simple foreign key. In the example, I will implement an entry mask that is table based, so it changes based on a related table's value.

I should warn you that calling a UDF has a great deal of overhead, and while you might get the urge to encapsulate a simple scalar for use in a CHECK constraint, it almost always isn't worth the overhead. As we have mentioned, CHECK constraints are executed once per row affected by the DML modification statement, and this extra cost will be compounded for every row affected by the modification query. I realize that this can be counterintuitive to a good programmer thinking that encapsulation is one of the most important goals of programming, but SQL is quite different from other types of programming in many ways because of the fact that you are pushing so much of the work to the engine, and the engine has its own way of doing things that you must respect.

Hence, it's best to try to express your Boolean expression without a UDF unless it's entirely necessary to access additional tables or do something more complex than a simple expression can. In the following examples, I'll employ UDFs to provide powerful rule checking, which can implement complex rules that would prove difficult to code using a simple Boolean expression.

You can implement the UDFs in either T-SQL or a .NET language (VB .NET, C#, or any .NET language that lets you exploit the capabilities of SQL Server 2005+ to write CLR-based objects in the database). In many cases, especially if you aren't doing any kind of table access in the code of the function, the CLR will perform much better than the T-SQL version.

As an example, I need to access values in a different table, so I'm going to build an example that implements an entry mask that varies based on the parent of a row. Consider that it's desirable to validate that catalog numbers for albums are of the proper format. However, different publishers have different catalog number masks for their clients' albums. (A more natural, yet distinctly more complex example would be phone numbers and addresses from around the world.)

For this example, I will continue to use the tables from the previous section. Note that the mask column, Publisher.CatalogNumberMask, needs to be considerably larger (five times larger in my example code) than the actual CatalogNumber column, because some of the possible masks use multiple characters to indicate a single character. You should also note that it's a varchar, even though the column is stored as a char value, because using char variables as LIKE masks can be problematic because of the space padding at the end of such columns (the comparison thinks that the extra space characters that are padded on the end of the fixed-length string need to match in the target string, which is rarely what's desired).

To do this, I build a T-SQL function that accesses this column to check that the value matches the mask, as shown (note that we'd likely build this constraint using T-SQL rather than by using the CLR, because it accesses a table in the body of the function):

```
CREATE FUNCTION Music.Publisher$CatalogNumberValidate
(
  @CatalogNumber char(12),
  @PublisherId int --not based on the Artist ID
)
```


www.it-ebooks.info

```
RETURNS bit
AS
BEGIN
    DECLARE @LogicalValue bit, @CatalogNumberMask varchar(100);

    SELECT @LogicalValue = CASE WHEN @CatalogNumber LIKE CatalogNumberMask
                                          THEN 1
                                        ELSE 0 END
    FROM Music.Publisher
    WHERE PublisherId = @PublisherId;

    RETURN @LogicalValue;
END;
```

When I loaded the data in the start of this section, I preloaded the data with valid values for the CatalogNumber and CatalogNumberMask columns:

```
SELECT Album.CatalogNumber, Publisher.CatalogNumberMask
FROM    Music.Album AS Album
            JOIN Music.Publisher AS Publisher
               ON Album.PublisherId = Publisher.PublisherId;
```

This returns the following results:

```
CatalogNumber   CatalogNumberMask
-------------   ------------------------------------------------------------
433-43ASD-33    [0-9][0-9][0-9]-[0-9][0-9][0-9a-z][0-9a-z][0-9a-z]-[0-9][0-9]
111-11111-11    [0-9][0-9][0-9]-[0-9][0-9][0-9a-z][0-9a-z][0-9a-z]-[0-9][0-9]
CD12345         [a-z][a-z][0-9][0-9][0-9][0-9][0-9]
```

Now, let's add the constraint to the table, as shown here:

```
ALTER TABLE Music.Album
  WITH CHECK ADD CONSTRAINT
    chkMusicAlbum$CatalogNumber$CatalogNumberValidate
    CHECK (Music.Publisher$CatalogNumbervalidate
                               (CatalogNumber,PublisherId) = 1);
```

If the constraint gives you errors because of invalid data existing in the table (because you were adding data, trying out the table, or in real development, this often occurs with test data from trying out the UI that they are building), you can use a query like the following to find them:

```
SELECT Album.Name, Album.CatalogNumber, Publisher.CatalogNumberMask
FROM    Music.Album AS Album
            JOIN Music.Publisher AS Publisher
                ON Publisher.PublisherId = Album.PublisherId
WHERE Music.Publisher$CatalogNumbervalidate(Album.CatalogNumber,Album.PublisherId) = 1;
```

Now, let's attempt to add a new row with an invalid value:

```
INSERT Music.Album(AlbumId, Name, ArtistId, PublisherId, CatalogNumber)
VALUES (4,'Who''s Next',2,2,'1');
```

This causes the error, because the catalog number of '1' doesn't match the mask set up for PublisherId number 2:

www.it-ebooks.info

```
Msg 547, Level 16, State 0, Line 1

The INSERT statement conflicted with the CHECK constraint
"chkMusicAlbum$CatalogNumber$CatalogNumberValidate". The conflict occurred in database
"Chapter7", table "Music.Album".
```

Now, changing change the catalog number to something that matches the entry mask the constraint is checking:

```
INSERT Music.Album(AlbumId, Name, ArtistId, CatalogNumber, PublisherId)
VALUES (4,'Who''s Next',2,'AC12345',2);

SELECT * FROM Music.Album;
```

This returns the following results, which you can see matches the '[a-z][a-z][0-9][0-9][0-9][0-9][0-9]' mask set up for the publisher with PublisherId = 2:

```
AlbumId     Name               ArtistId     CatalogNumber    PublisherId
-------     ---------------    --------     -------------    -----------
1           The White Album    1            433-43ASD-33     1
2           Revolver           1            111-11111-11     1
3           Quadrophenia       2            CD12345          2
4           Who's Next         2            AC12345          2
```

Using this kind of approach, you can build any single-row validation code for your tables. As described previously, each UDF will fire once for each row and each column that was modified in the update. If you are making large numbers of inserts, performance might suffer, but having data that you can trust is worth it.

We will talk about triggers later in this chapter, but alternatively, you could create a trigger that checks for the existence of any rows returned by a query, based on the query used earlier to find improper data in the table:

```
SELECT *
FROM    Music.Album AS Album
            JOIN Music.Publisher AS Publisher
                ON Publisher.PublisherId = Album.PublisherId
WHERE Music.Publisher$CatalogNumberValidate
    (Album.CatalogNumber, Album.PublisherId) <> 1;
```

There's one drawback to this type of constraint, whether implemented in a constraint or trigger. As it stands right now, the `Album` table is protected from invalid values being entered into the `CatalogNumber` column, but it doesn't say anything about what happens if a user changes the `CatalogEntryMask` on the `Publisher` table. If this is a concern, you'd need to add a `CHECK` constraint to the `Publisher` table that validates changes to the mask against any existing data.

■ **Caution** Using user-defined functions that access other rows in the same table is dangerous, because while the data for each row appears in the table as the function is executed, if multiple rows are updated simultaneously, those rows do not appear to be in the table, so if an error condition exists only in the rows that are being modified, your final results could end up in error.

www.it-ebooks.info

Enhancing Errors Caused by Constraints

The real downside to check constraints is the error messages they produce upon failure. The error messages are certainly things you don't want to show to a user, if for no other reason other than they will generate help desk calls every time typical users see them. Dealing with these errors is one of the more annoying parts of using constraints in SQL Server.

Whenever a statement fails a constraint requirement, SQL Server provides you with an ugly message and offers no real method for displaying a clean message automatically. Luckily, SQL Server 2005 implemented vastly improved error-handling capabilities in T-SQL over previous versions. In this section, I'll briefly detail a way to refine the ugly messages you get from a constraint error message, much like the error from the previous statement:

```
Msg 547, Level 16, State 0, Line 1

The INSERT statement conflicted with the CHECK constraint
"chkMusicAlbum$CatalogNumber$CatalogNumberValidate". The conflict occurred in database
"Chapter7", table "Music.Album".
```

I'll show you how to map this to an error message that at least makes some sense. First, the parts of the error message are as follows:

- *Error number*—`Msg 547`: The error number that's passed back to the calling program. In some cases, this error number is significant; however, in most cases it's enough to say that the error number is nonzero.
- *Level*—`Level 16`: A severity level for the message. `0` through `18` are generally considered to be user messages, with `16` being the default. Levels `19`–`25` are severe errors that cause the connection to be severed (with a message written to the log) and typically involve data corruption issues.
- *State*—`State 0`: A value from `0`–`127` that represents the state of the process when the error was raised. This value is rarely used by any process.
- *Line*—`Line 1`: The line in the batch or object where the error is occurring. This value can be extremely useful for debugging purposes.
- *Error description*: A text explanation of the error that has occurred.

In its raw form, this is the exact error that will be sent to the client. Using the new `TRY-CATCH` error handling, we can build a simple error handler and a scheme for mapping constraints to error messages (or you can do much the same thing in client code as well for errors that you just cannot prevent from your user interface). Part of the reason we name constraints is to determine what the intent was in creating the constraint in the first place. In the following code, we'll implement a very rudimentary error-mapping scheme by parsing the text of the name of the constraint from the message, and then we'll look this value up in a mapping table. It isn't a "perfect" scheme, but it does the trick when using constraints as the only data protection for a situation (it is also helps you to document the errors that your system may raise as well).

First, let's create a mapping table where we put the name of the constraint that we've defined and a message that explains what the constraint means:

```
CREATE SCHEMA utility; --used to hold objects for utility purposes
GO
CREATE TABLE utility.ErrorMap
(
  ConstraintName sysname NOT NULL PRIMARY KEY,
  Message varchar(2000) NOT NULL
);
```


www.it-ebooks.info

```
GO
INSERT utility.ErrorMap(constraintName, message)
VALUES ('chkMusicAlbum$CatalogNumber$CatalogNumberValidate',
        'The catalog number does not match the format set up by the Publisher');
```

Then, we create a procedure to do the actual mapping by taking the values that can be retrieved from the ERROR_%() procedures that are accessible in a CATCH block and using them to look up the value in the ErrorMap table:

```
CREATE PROCEDURE utility.ErrorMap$MapError
(

  @ErrorNumber int = NULL,
  @ErrorMessage nvarchar(2000) = NULL,
  @ErrorSeverity INT= NULL

) AS
  BEGIN
    SET NOCOUNT ON

  --use values in ERROR_ functions unless the user passes in values
  SET @ErrorNumber = COALESCE(@ErrorNumber, ERROR_NUMBER());
  SET @ErrorMessage = COALESCE(@ErrorMessage, ERROR_MESSAGE());
  SET @ErrorSeverity = COALESCE(@ErrorSeverity, ERROR_SEVERITY());

  --strip the constraint name out of the error message
  DECLARE @constraintName sysname;
  SET @constraintName = SUBSTRING( @ErrorMessage,
      CHARINDEX('constraint "',@ErrorMessage) + 12,
      CHARINDEX('"',substring(@ErrorMessage,
      CHARINDEX('constraint "',@ErrorMessage) + 12,2000))-1)

  --store off original message in case no custom message found
  DECLARE @originalMessage nvarchar(2000);
  SET @originalMessage = ERROR_MESSAGE();

  IF @ErrorNumber = 547 --constraint error
    BEGIN
      SET @ErrorMessage =
            (SELECT message
            FROM utility.ErrorMap
            WHERE constraintName = @constraintName
            );
    END
  --if the error was not found, get the original message with generic 50000 error number
    SET @ErrorMessage = ISNULL(@ErrorMessage, @originalMessage);
    THROW 50000, @ErrorMessage, @ErrorSeverity;
  END
GO
```


www.it-ebooks.info

Now, see what happens when we enter an invalid value for an album catalog number:

```
BEGIN TRY
  INSERT Music.Album(AlbumId, Name, ArtistId, CatalogNumber, PublisherId)
  VALUES (5,'who are you',2,'badnumber',2);
END TRY
BEGIN CATCH
  EXEC utility.ErrorMap$MapError;
END CATCH
```

The error message is as follows:

```
Msg 50000, Level 16, State 1, Procedure ErrorMap$mapError, Line 24
The catalog number does not match the format set up by the Publisher
```

rather than:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint
"chkMusicAlbum$CatalogNumber$CatalogNumberValidate". The conflict occurred in database
"Chapter7", table "Music.Album".
```

This is far more pleasing, even if it was a bit of a workout getting to this new message.

DML Triggers

Triggers are a type of stored procedure attached to a table or view that is executed automatically when the contents of a table are changed. While they share the ability to enforce data protection, they differ from constraints in being far more flexible because you can code them like stored procedures and you can introduce side effects like formatting input data or cascading an operation to another table. You can use them to enforce almost any business rule, and they're especially important for dealing with situations that are too complex for a CHECK constraint to handle. We used triggers in the previous chapter to automatically manage row update date values.

Triggers often get a bad name because they can be pretty quirky, especially because they can kill performance when you are dealing with large updates. For example, if you have a trigger on a table and try to update a million rows, you are likely to have issues. However, for most OLTP operations in a relational database, operations shouldn't be touching more than a handful of rows at a time. Trigger usage does need careful consideration, but where they are needed, they are terribly useful. My recommendation is to use triggers when you need to do the following:

- Perform cross-database referential integrity.
- Check inter-row rules, where just looking at the current row isn't enough for the constraints.
- Check inter-table constraints, when rules require access to data in a different table.
- Introduce desired side effects to your data-modification queries, such as maintaining required denormalizations.

www.it-ebooks.info

- Guarantee that no insert, update, or delete operations can be executed on a table, even if the user does have rights to perform the operation.

Some of these operations could also be done in an application layer, but for the most part, these operations are far easier and safer (particularly for data integrity) when done automatically using triggers. When it comes to data protection, the primary advantages that triggers have over constraints is the ability to access other tables seamlessly and to operate on multiple rows at once. In Appendix B, I will discuss a bit more of the mechanics of writing triggers and their limitations. In this chapter, I am going to create DML triggers to handle typical business needs.

There are two different types of DML triggers that we will make use of in this section. Each type can be useful in its own way, but they are quite different in why they are used.

- `AFTER`: These triggers fire after the DML statement (`INSERT`/`UPDATE`/`DELETE`) has affected the table. `AFTER` triggers are usually used for handling rules that won't fit into the mold of a constraint, for example, rules that require data to be stored, such as a logging mechanism. You may have a virtually unlimited number of `AFTER` triggers that fire on `INSERT`, `UPDATE`, and `DELETE`, or any combination of them.
- `INSTEAD OF`: These triggers operate "instead of" the built-in command (`INSERT`, `UPDATE`, or `DELETE`) affecting the table or view. In this way, you can do whatever you want with the data, either doing exactly what was requested by the user or doing something completely different (you can even just ignore the operation altogether). You can have a maximum of one `INSTEAD OF INSERT`, `UPDATE`, and `DELETE` trigger of each type per table. It is allowed (but not a generally good idea) to combine all three into one and have a single trigger that fires for all three operations.

This section will be split between these two types of triggers because they have two very different sets of use cases. Since coding triggers is not one of the more well trod topics in SQL Server, in Appendix B, I will introduce trigger coding techniques and provide a template that we will use throughout this chapter (it's the template we used in Chapter 6, too).

AFTER Triggers

`AFTER` triggers fire after the DML statement has completed. They are the most common trigger that people use, because they have the widest array of uses. Though triggers may not seem very useful, back in SQL Server 6.0 and earlier, there were no `CHECK` constraints, and even `FOREIGN KEYS` where just being introduced, so all data protection was managed using triggers. Other than being quite cumbersome to maintain, some fairly complex systems were created using hardware that is comparable to one of my Logitech Harmony remote controls.

In this section on `AFTER` triggers, I will present examples that demonstrate several forms of triggers that I use to solve problems that are reasonably common. I'll give examples of the following usages of triggers:

- Range checks on multiple rows
- Maintaining summary values (only as necessary)
- Cascading inserts
- Child-to-parent cascades
- Maintaining an audit trail
- Relationships that span databases and servers

From these examples, you should be able to extrapolate almost any use of `AFTER` triggers. Just keep in mind that, although triggers are not the worst thing for performance, they should be used no more than necessary.

www.it-ebooks.info

■ **Note** For one additional example, check the section on uniquness in Chapter 8, where I will implement a type of uniqueness based on ranges of data using a trigger-based solution.

Range Checks on Multiple Rows

The first type of check we'll look at is the range check, in which we want to make sure that a column is within some specific range of values. You can do range checks using a CHECK constraint to validate the data in a single row (for example, `column > 10`) quite easily. However, you wouldn't want to use them to validate conditions based on aggregates of multiple rows (`sum(column) > 10`), because if you updated 100 rows, you would have to do 100 validations where one statement could do the same work.

If you need to check that a row or set of rows doesn't violate a given condition, usually based on an aggregate like a maximum sum, you should use a trigger. As an example, I'll look at a simple accounting system. As users deposit and withdraw money from accounts, you want to make sure that the balances never dip below zero. All transactions for a given account have to be considered.

First, we create a schema for the accounting groups:

```
CREATE SCHEMA Accounting;
```

Then, we create a table for an account and then one to contain the activity for the account:

```
CREATE TABLE Accounting.Account
(
  AccountNumber char(10)
    constraint PKAccounting_Account PRIMARY KEY
  --would have other columns
);

CREATE TABLE Accounting.AccountActivity
(
  AccountNumber char(10) NOT NULL
    constraint Accounting_Account$has$Accounting_AccountActivity
    foreign key references Accounting.Account(AccountNumber),
  --this might be a value that each ATM/Teller generates
  TransactionNumber     char(20) NOT NULL,
  Date                  datetime2(3) NOT NULL,
  TransactionAmount     numeric(12,2) NOT NULL,
  constraint PKAccounting_AccountActivity
    PRIMARY KEY (AccountNumber, TransactionNumber)
);
```

Now, we add a trigger to the `Accounting.AccountActivity` table that checks to make sure that when you sum together the transaction amounts for an `Account`, that the sum is greater than zero:

```
CREATE TRIGGER Accounting.AccountActivity$insertUpdateTrigger
ON Accounting.AccountActivity
AFTER INSERT,UPDATE AS
```


www.it-ebooks.info

```
BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  -- @rowsAffected int = (SELECT COUNT(*) FROM deleted);
  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  BEGIN TRY

  --[validation section]
  --disallow Transactions that would put balance into negatives
  IF EXISTS ( SELECT AccountNumber
              FROM Accounting.AccountActivity AS AccountActivity
              WHERE EXISTS (SELECT *
                            FROM inserted
                            WHERE inserted.AccountNumber =
                              AccountActivity.AccountNumber)
              GROUP BY AccountNumber
              HAVING SUM(TransactionAmount) < 0)

     BEGIN
       IF @rowsAffected = 1
           SELECT @msg = 'Account: ' + AccountNumber +
                  ' TransactionNumber:' +
                  CAST(TransactionNumber as varchar(36)) +
                  ' for amount: ' + CAST(TransactionAmount as varchar(10))+
                  ' cannot be processed as it will cause a negative balance'
           FROM inserted;
       ELSE
         SELECT @msg = 'One of the rows caused a negative balance';
         THROW 50000, @msg, 16;
     END

  --[modification section]
  END TRY
  BEGIN CATCH
            IF @@trancount > 0
                ROLLBACK TRANSACTION;

            THROW; --will halt the batch or be caught by the caller's catch block
  END CATCH

END;
```


www.it-ebooks.info

The key to using this type of trigger is to look for the existence of rows in the base table, not the rows in the inserted table, because the concern is how the inserted rows affect the overall status for an Account. Take this query, which we'll use to determine whether there are rows that fail the criteria:

```
SELECT AccountNumber
FROM Accounting.AccountActivity AS AccountActivity
WHERE EXISTS (SELECT *
              FROM inserted
              WHERE inserted.AccountNumber = AccountActivity.AccountNumber)

GROUP BY AccountNumber
HAVING SUM(TransactionAmount) < 0;
```

The key here is that we could remove the bold part of the query, and it would check all rows in the table. The WHERE clause simply makes sure that the only rows we consider are for accounts that have new data inserted. This way, we don't end up checking all rows that we know our query hasn't touched. Note, too, that I don't use a JOIN operation. By using an EXISTS criteria in the WHERE clause, we don't affect the cardinality of the set being returned in the FROM clause, no matter how many rows in the inserted table have the same AccountNumber.

To see it in action, use this code:

```
--create some set up test data
INSERT INTO Accounting.Account(AccountNumber)
VALUES ('1111111111');

INSERT INTO Accounting.AccountActivity(AccountNumber, TransactionNumber,
        Date, TransactionAmount)
VALUES ('1111111111','A00000000000000000001','20050712',100),
       ('1111111111','A00000000000000000002','20050713',100);
```

Now, let's see what happens when we violate this rule:

```
INSERT INTO Accounting.AccountActivity(AccountNumber, TransactionNumber,
        Date, TransactionAmount)
VALUES ('1111111111','A00000000000000000003','20050713',-300);
```

Here's the result:

```
Msg 50000, Level 16, State 16, Procedure AccountActivity$insertUpdateTrigger, Line 36

Account: 1111111111 TransactionNumber:A00000000000000000003 for amount: -300.00 cannot be
processed as it will cause a negative balance
```

The error message is the custom error message that we coded in the case where a single row was modified. Now, let's make sure that the trigger works when we have greater than one row in the INSERT statement:

```
--create new Account
INSERT INTO Accounting.Account(AccountNumber)
VALUES ('2222222222');
GO
--Now, this data will violate the constraint for the new Account:
INSERT INTO Accounting.AccountActivity(AccountNumber, TransactionNumber,
        Date, TransactionAmount)
```


www.it-ebooks.info

```
VALUES  ('1111111111','A0000000000000000004','20050714',100),
        ('2222222222','A0000000000000000005','20050715',100),
        ('2222222222','A0000000000000000006','20050715',100),
        ('2222222222','A0000000000000000007','20050715',-201);
```

This causes the following error:

```
Msg 50000, Level 16, State 16, Procedure AccountActivity$insertUpdateTrigger, Line 36
One of the rows caused a negative balance
```

The multirow error message is much less informative, though you could expand it to include information about a row (or all the rows) that caused the violation with some more text, even showing the multiple failed values. Usually a simple message is sufficient to deal with, because generally if multiple rows are being modified in a single statement, it's a batch process, and the complexity of building error messages is way more than it's worth. Processes would likely be established on how to deal with certain errors being returned.

■ **Tip** In the error message, note that the first error states it's from line 36. This is line 36 of the trigger where the error message was raised. This can be valuable information when debugging triggers (and any SQL code, really). Note also that because the `ROLLBACK` command was used in the trigger, the batch will be terminated. This will be covered in more detail in the "Dealing with Triggers and Constraints Errors" section later in this chapter.

VIEWING TRIGGER EVENTS

To see the events for which a trigger fires, you can use the following query:

```
SELECT trigger_events.type_desc
FROM sys.trigger_events
    JOIN sys.triggers
      ON sys.triggers.object_id = sys.trigger_events.object_id
WHERE triggers.name = 'AccountActivity$insertUpdateTrigger';
```

This returns `INSERT` and `UPDATE` in two rows, because we declared the `Accounting.AccountActivity$insertUpdateTrigger` trigger to fire on `INSERT` and `UPDATE` operations.

Maintaining Summary Values

Maintaining summary values is generally not necessary, and doing so typically is just a matter of poor normalization or perhaps a misguided attempt to optimize where a better database design would have sufficed. However, there are cases where some form of active summarization may be necessary:

- There is no other reasonable method available.
- The amount of data to be summarized is large.
- The amount of reads of the summary values is far greater than the activity on the lower values.

www.it-ebooks.info

As an example, let's extend the previous example of the Account and AccountActivity tables from the "Range Checks on Multiple Rows" section. To the Account table, I will add a BalanceAmount column:

```
ALTER TABLE Accounting.Account
  ADD BalanceAmount numeric(12,2) NOT NULL
    CONSTRAINT DfltAccounting_Account_BalanceAmount DEFAULT (0.00);
```

Then, we will update the Balance column to have the current value of the data in the -AccountActivity rows. First, running this query to view the expected values:

```
SELECT Account.AccountNumber,
       SUM(coalesce(AccountActivity.TransactionAmount,0.00)) AS NewBalance
FROM Accounting.Account
       LEFT OUTER JOIN Accounting.AccountActivity
          ON Account.AccountNumber = AccountActivity.AccountNumber
GROUP BY Account.AccountNumber;
```

This returns the following:

```
AccountNumber   NewBalance
-------------   ----------
1111111111      200.00
2222222222        0.00
```

Now, update the BalanceAmount column values to the existing rows using the following statement:

```
WITH Updater AS (
SELECT Account.AccountNumber,
       SUM(coalesce(TransactionAmount,0.00)) AS NewBalance
FROM Accounting.Account
       LEFT OUTER JOIN Accounting.AccountActivity
          On Account.AccountNumber = AccountActivity.AccountNumber
GROUP BY Account.AccountNumber)
UPDATE Account
SET BalanceAmount = Updater.NewBalance
FROM Accounting.Account
     JOIN Updater
        ON Account.AccountNumber = Updater.AccountNumber;
```

That statement will make the basis of our changes to the trigger that we added in the previous section (the changes appear in bold). The only change that needs to be made is to filter the Account set down to the accounts that were affected by the DML that cause the trigger to fire. Using an EXISTS filter lets you not have to worry about whether one new row was created for the account or 100.

```
ALTER TRIGGER Accounting.AccountActivity$insertUpdateTrigger
ON Accounting.AccountActivity
AFTER INSERT,UPDATE AS
BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
```


www.it-ebooks.info

```
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
     @rowsAffected int = (SELECT COUNT(*) FROM inserted);
      -- @rowsAffected int = (SELECT COUNT(*) FROM deleted);

BEGIN TRY

  --[validation section]
  --disallow Transactions that would put balance into negatives
  IF EXISTS ( SELECT AccountNumber
              FROM Accounting.AccountActivity as AccountActivity
              WHERE EXISTS (SELECT *
                            FROM inserted
                            WHERE inserted.AccountNumber =
                                  AccountActivity.AccountNumber)
  GROUP BY AccountNumber
  HAVING sum(TransactionAmount) < 0)

BEGIN

  IF @rowsAffected = 1
    SELECT @msg = 'Account: ' + AccountNumber +
      ' TransactionNumber:' +
     cast(TransactionNumber AS varchar(36)) +
     ' for amount: ' + cast(TransactionAmount as varchar(10))+
     ' cannot be processed as it will cause a negative balance'
    FROM inserted;
  ELSE
    SELECT @msg = 'One of the rows caused a negative balance';
    THROW 50000, @msg, 16;

END

--[modification section]
IF UPDATE (TransactionAmount)
  BEGIN
    ;WITH Updater AS (
  SELECT Account.AccountNumber,
    SUM(coalesce(TransactionAmount,0.00)) AS NewBalance
  FROM Accounting.Account
    LEFT OUTER JOIN Accounting.AccountActivity
        On Account.AccountNumber = AccountActivity.AccountNumber
    --This where clause limits the summarizations to those rows
    --that were modified by the DML statement that caused
    --this trigger to fire.
  WHERE EXISTS (SELECT *
                FROM Inserted
                WHERE Account.AccountNumber = Inserted.AccountNumber)
  GROUP BY Account.AccountNumber)
```


www.it-ebooks.info

```
    UPDATE Account
    SET BalanceAmount = Updater.NewBalance
    FROM Accounting.Account
        JOIN Updater
              ON Account.AccountNumber = Updater.AccountNumber;
      END

  END TRY
    BEGIN CATCH
      IF @@trancount > 0
        ROLLBACK TRANSACTION;
      THROW; --will halt the batch or be caught by the caller's catch block
    END CATCH

  END;
```

Now, insert a new row into AccountActivity:

```
INSERT INTO Accounting.AccountActivity(AccountNumber, TransactionNumber,
        Date, TransactionAmount)
VALUES ('1111111111','A0000000000000000004','20050714',100);
```

Next, examine the state of the Account table, comparing it to the query used previously to check what the balances should be:

```
SELECT Account.AccountNumber,Account.BalanceAmount,
        SUM(COALESCE(AccountActivity.TransactionAmount,0.00)) AS SummedBalance
FROM Accounting.Account
        LEFT OUTER JOIN Accounting.AccountActivity
          ON Account.AccountNumber = AccountActivity.AccountNumber
GROUP BY Account.AccountNumber,Account.BalanceAmount;
```

which returns the following, showing that the sum is the same as the stored balance:

```
AccountNumber    BalanceAmount      SummedBalance
-------------    -------------      -------------
1111111111       300.00             300.00
2222222222       0.00               0.00
```

The next step—the multirow test—is very important when building a trigger such as this. You need to be sure that if a user inserts more than one row at a time, it will work. In our example, we will insert rows for both accounts in the same DML statement and two rows for one of the accounts. This is not a sufficient test necessarily, but it's enough for demonstration purposes at least:

```
INSERT INTO Accounting.AccountActivity(AccountNumber, TransactionNumber,
        Date, TransactionAmount)
VALUES  ('1111111111','A0000000000000000005','20050714',100),
        ('2222222222','A0000000000000000006','20050715',100),
        ('2222222222','A0000000000000000007','20050715',100);
```


www.it-ebooks.info

Again, the query on the AccountActivity and Account should show the same balances:

```
AccountNumber   BalanceAmount     SummedBalance
-------------   -------------     -------------
1111111111      400.00            400.00
2222222222      200.00            200.00
```

If you wanted a DELETE trigger (and in the case of a ledger like this, you generally do not want to actually delete rows but rather insert offsetting values, so to delete a $100 insert, you would insert a –100), the only difference is that instead of the EXISTS condition referring to the inserted table, it needs to refer to the deleted table:

```
CREATE TRIGGER Accounting.AccountActivity$deleteTrigger
ON Accounting.AccountActivity
AFTER DELETE AS
BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
   -- @rowsAffected int = (SELECT COUNT(*) FROM inserted);
      @rowsAffected int = (SELECT COUNT(*) FROM deleted);

  BEGIN TRY

  --[validation section]
  --[modification section]
    ;WITH Updater as (
        SELECT Account.AccountNumber,
             SUM(COALESCE(TransactionAmount,0.00)) as NewBalance
        FROM Accounting.Account
             LEFT OUTER JOIN Accounting.AccountActivity
                On Account.AccountNumber = AccountActivity.AccountNumber
        WHERE EXISTS (SELECT *
                      FROM deleted
                      WHERE Account.AccountNumber =
                            deleted.AccountNumber)
        GROUP BY Account.AccountNumber, Account.BalanceAmount)
        UPDATE Account
        SET BalanceAmount = Updater.NewBalance
        FROM Accounting.Account
             JOIN Updater
                   ON Account.AccountNumber = Updater.AccountNumber;

 END TRY
```


www.it-ebooks.info

```
 BEGIN CATCH
      IF @@trancount > 0
        ROLLBACK TRANSACTION;
      THROW; --will halt the batch or be caught by the caller's catch block
 END CATCH

END;
GO
```

So now, delete a couple of transactions:

```
DELETE Accounting.AccountActivity
WHERE TransactionNumber IN ('A0000000000000000004',
                            'A0000000000000000005');
```

Checking the balance now, you will see that the balance for Account 1111111111 has been decremented to 200:

```
AccountNumber    BalanceAmount    SummedBalance
-------------    -------------    -------------
1111111111       200.00           200.00
2222222222       200.00           200.00
```

I can't stress enough that this type of strategy should be the exception, not the rule. But when you have to implement summary data, using a trigger is the way to go in most cases. One of the more frustrating things to have to deal with is summary data that is out of whack, because it takes time away from making progress with creating new software.

■ **Caution** I want to reiterate to be extremely careful to test your code extra thoroughly when you include denormalizations like this. If you have other DML in triggers that insert or update into the same table, there is a chance that the trigger will not fire again, based on how you have the nested triggers and recursive triggers options set that I discussed previously. Good testing strategies are important in all cases really, but the point here is to be extra careful when using triggers to modify data.

Cascading Inserts

A cascading insert refers to the situation whereby after a row is inserted into a table, one or more other new rows are automatically inserted into other tables. This is frequently done when you need to initialize a row in another table, quite often a status of some sort.

For this example, we're going to build a small system to store URLs for a website-linking system. During low-usage periods, an automated browser connects to the URLs so that they can be verified (hopefully, limiting broken links on web pages).

To implement this, I'll use the set of tables in Figure 7-3.

www.it-ebooks.info

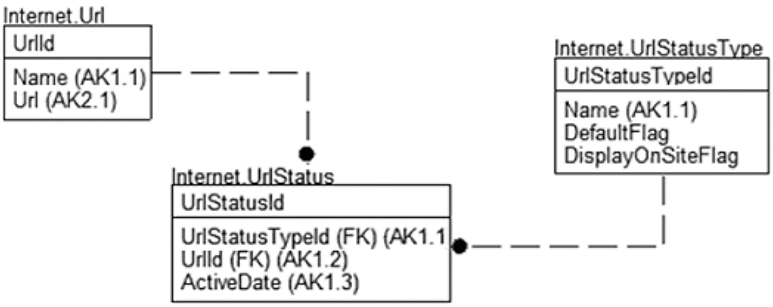


Figure 7-3. *Storing URLs for a website-linking system*

```
CREATE SCHEMA Internet;
GO
CREATE TABLE Internet.Url
(
    UrlId int NOT NULL identity(1,1) constraint PKUrl primary key,
    Name varchar(60) NOT NULL constraint AKInternet_Url_Name UNIQUE,
    Url varchar(200) NOT NULL constraint AKInternet_Url_Url UNIQUE
);

--Not a user manageable table, so not using identity key (as discussed in
--Chapter 5 when I discussed choosing keys) in this one table. Others are
--using identity-based keys in this example.
CREATE TABLE Internet.UrlStatusType
(
    UrlStatusTypeId int NOT NULL
     CONSTRAINT PKInternet_UrlStatusType PRIMARY KEY,
    Name varchar(20) NOT NULL
     CONSTRAINT AKInternet_UrlStatusType UNIQUE,
    DefaultFlag bit NOT NULL,
    DisplayOnSiteFlag bit NOT NULL
);

CREATE TABLE Internet.UrlStatus
(
    UrlStatusId int NOT NULL IDENTITY(1,1)
       CONSTRAINT PKInternet_UrlStatus PRIMARY KEY,
    UrlStatusTypeId int NOT NULL
       CONSTRAINT
 Internet_UrlStatusType$defines_status_type_of$Internet_UrlStatus
    REFERENCES Internet.UrlStatusType(UrlStatusTypeId),
    UrlId int NOT NULL
      CONSTRAINT Internet_Url$has_status_history_in$Internet_UrlStatus
       REFERENCES Internet.Url(UrlId),
    ActiveTime datetime2(3) NOT NULL,
    CONSTRAINT AKInternet_UrlStatus_statusUrlDate
       UNIQUE (UrlStatusTypeId, UrlId, ActiveTime)
);
```


www.it-ebooks.info

```
--set up status types
INSERT Internet.UrlStatusType (UrlStatusTypeId, Name,
               DefaultFlag, DisplayOnSiteFlag)
VALUES  (1, 'Unverified',1,0),
        (2, 'Verified',0,1),
        (3, 'Unable to locate',0,0);
```

The Url table holds URLs to different sites on the Web. When someone enters a URL, we initialize the status to 'Unverified'. A process should be in place in which the site is checked often to make sure nothing has changed (particularly the unverified ones!).

You begin by building a trigger that inserts a row into the UrlStatus table on an insert that creates a new row with the UrlId and the default UrlStatusType based on DefaultFlag having the value of 1.

```
CREATE TRIGGER Internet.Url$afterInsertTrigger
ON Internet.Url
AFTER INSERT AS
BEGIN

  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
     @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  -- @rowsAffected int = (SELECT COUNT(*) FROM deleted);

  BEGIN TRY

    --[validation section]
    --[modification section]
    --add a row to the UrlStatus table to tell it that the new row
    --should start out as the default status
    INSERT INTO Internet.UrlStatus (UrlId, UrlStatusTypeId, ActiveTime)
    SELECT inserted.UrlId, UrlStatusType.UrlStatusTypeId,
      SYSDATETIME()
    FROM inserted
       CROSS JOIN (SELECT UrlStatusTypeId
                   FROM UrlStatusType
                   WHERE DefaultFlag = 1) as UrlStatusType;
                     --use cross join with a WHERE clause
                     --as this is not technically a join
                     --between inserted and UrlType
    END TRY
    BEGIN CATCH
    IF @@trancount > 0
      ROLLBACK TRANSACTION;
    THROW; --will halt the batch or be caught by the caller's catch block
    END CATCH
END;
```


www.it-ebooks.info

The idea here is that for every row in the inserted table, we'll get the single row from the `UrlStatusType` table that has `DefaultFlag` equal to `1`. So, let's try it:

```
INSERT Internet.Url(Name, Url)
VALUES  ('More info can be found here',
        'http://sqlblog.com/blogs/louis_davidson/default.aspx');
SELECT * FROM Internet.Url;
SELECT * FROM Internet.UrlStatus;
```

This returns the following results:

```
UrlId       Name                          Url
-----       ----------------------------  ----------------------------------------------------
1           More info can be found here   http://sqlblog.com/blogs/louis_davidson/default.aspx

UrlStatusId     UrlStatusTypeId      UrlId          ActiveTime
-----------     ---------------      -----          ----------------------
1               1                    1              2011-06-10 00:11:40.480
```

■ **Tip** It's easier if users can't modify tables such as the `UrlStatusType` table, so there cannot be a case where there's no status set as the default (or too many rows). If there were no default status, the URL would never get used, because the processes wouldn't see it. You could also create a trigger to check to see whether more than one row is set to the default, but the trigger still doesn't protect you against there being zero rows that are set to the default.

Cascading from Child to Parent

All the cascade operations that you can do with constraints (`CASCADE` or `SET NULL`) are strictly from parent to child. Sometimes, you want to go the other way around and delete the parents of a row when you delete the child. Typically, you do this when the child is what you're interested in and the parent is simply maintained as an attribute of the child. Also typical of this type of situation is that you want to delete the parent only if all children are deleted.

In our example, we have a small model of my game collection. I have several game systems and quite a few games. Often, I have the same game on multiple platforms, so I want to track this fact, especially if I want to trade a game that I have on multiple platforms for something else. So, we have a table for the `GamePlatform` (the system) and another for the actual game itself. This is a many-to-many relationship, so we have an associative entity called `GameInstance` to record ownership, as well as when the game was purchased for the given platform. Each of these tables has a delete-cascade relationship, so all instances are removed. What about the games, though? If all `GameInstance` rows are removed for a given game, we want to delete the game from the database. The tables are shown in Figure 7-4.

www.it-ebooks.info

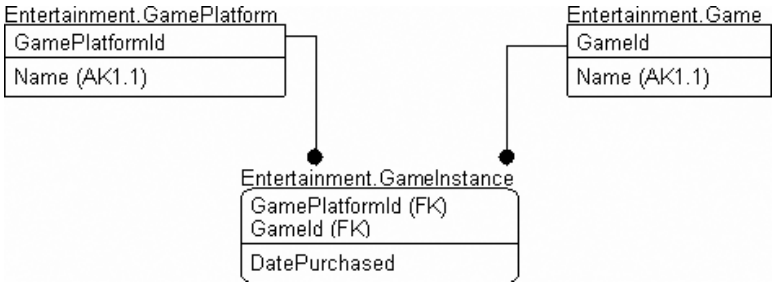


Figure 7-4. *The game tables*

```
--start a schema for entertainment-related tables

CREATE SCHEMA Entertainment;
GO
CREATE TABLE Entertainment.GamePlatform
(
  GamePlatformId int NOT NULL CONSTRAINT PKEntertainmentGamePlatform PRIMARY KEY
  Name varchar(50) NOT NULL CONSTRAINT AKEntertainmentGamePlatform_Name UNIQUE
);
CREATE TABLE Entertainment.Game
(
  GameId int NOT NULL CONSTRAINT PKEntertainmentGame PRIMARY KEY,
  Name   varchar(50) NOT NULL CONSTRAINT AKEntertainmentGame_Name UNIQUE
  --more details that are common to all platforms
);

--associative entity with cascade relationships back to Game and GamePlatform
CREATE TABLE Entertainment.GameInstance
(
  GamePlatformId int NOT NULL,
  GameId int NOT NULL,
  PurchaseDate date NOT NULL,
  CONSTRAINT PKEntertainmentGameInstance PRIMARY KEY (GamePlatformId, GameId),
  CONSTRAINT
  EntertainmentGame$is_owned_on_platform_by$EntertainmentGameInstance
     FOREIGN KEY (GameId) REFERENCES Entertainment.Game(GameId)
         ON DELETE CASCADE,
  CONSTRAINT
      EntertainmentGamePlatform$is_linked_to$EntertainmentGameInstance
  FOREIGN KEY (GamePlatformId)
      REFERENCES Entertainment.GamePlatform(GamePlatformId)
      ON DELETE CASCADE
);
```


www.it-ebooks.info

Then, I insert a sampling of data:

```
INSERT into Entertainment.Game (GameId, Name)
VALUES  (1,'Lego Pirates of the Carribean').
        (2,'Legend Of Zelda: Ocarina of Time');

INSERT into Entertainment.GamePlatform(GamePlatformId, Name)
VALUES  (1,'Nintendo Wii'), --Yes, as a matter of fact I am still a
        (2,'Nintendo 3DS'); --Nintendo Fanboy, why do you ask?

INSERT into Entertainment.GameInstance(GamePlatformId, GameId, PurchaseDate)
VALUES  (1,1,'20110804'),
        (1,2,'20110810'),
        (2,2,'20110604');

--the full outer joins ensure that all rows are returned from all sets, leaving
--nulls where data is missing
SELECT GamePlatform.Name as Platform, Game.Name as Game, GameInstance. PurchaseDate
FROM Entertainment.Game as Game
    FULL OUTER JOIN Entertainment.GameInstance as GameInstance
      ON Game.GameId = GameInstance.GameId
    Full OUTER Join Entertainment.GamePlatform
      ON GamePlatform.GamePlatformId = GameInstance.GamePlatformId;
```

As you can see, I have two games for Wii and only a single one for Nintendo 3DS:

```
Platform         Game                 PurchaseDate
------------     ------------------   ------------
Nintendo Wii     Lego Star Wars III   2011-08-04
Nintendo Wii     Ocarina of Time      2011-08-10
Nintendo 3DS     Ocarina of Time      2011-06-04
```

So, we create a trigger on the table to do the "reverse" cascade operation:

```
CREATE TRIGGER Entertainment.GameInstance$afterDeleteTrigger
ON Entertainment.GameInstance
AFTER DELETE AS

BEGIN

  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  -- @rowsAffected int = (SELECT COUNT(*) FROM inserted);
     @rowsAffected int = (SELECT COUNT(*) FROM deleted);
```


www.it-ebooks.info

```
  BEGIN TRY

  --[validation section]
  --[modification section]

    --delete all Games
    DELETE Game --where the GameInstance was deleted
    WHERE GameId in (SELECT deleted.GameId
      FROM deleted --and there are no GameInstances
      WHERE not exists (SELECT * --left
                  FROM GameInstance
                  WHERE GameInstance.GameId =
                        deleted.GameId));
  END TRY
  BEGIN CATCH
     IF @@trancount > 0
       ROLLBACK TRANSACTION;
     THROW; --will halt the batch or be caught by the caller's catch block
  END CATCH
END
```

It's as straightforward as that. Just delete the games, and let the trigger cover the rest. Delete the row for the Wii:

```
DELETE Entertainment.GamePlatform
WHERE  GamePlatformId = 1;
```

Next, check the data

```
SELECT GamePlatform.Name AS Platform, Game.Name AS Game, GameInstance. PurchaseDate
FROM   Entertainment.Game AS Game
       FULL OUTER JOIN Entertainment.GameInstance AS GameInstance
         ON Game.GameId = GameInstance.GameId
       FULL OUTER JOIN Entertainment.GamePlatform
         ON GamePlatform.GamePlatformId = GameInstance.GamePlatformId;
```

You can see that now we have only a single row in the Game table:

```
platform        Game                PurchaseDate
------------    ------------------  ------------
Nintendo 3DS    Ocarina of Time 3D  2011-06-04
```

Maintaining an Audit Trail

A common task that's implemented using triggers is the audit trail or audit log. You use it to record previous versions of rows or columns so you can determine who changed a given row. Often, an audit trail is simply for documentation purposes, so we can go back to other users and ask why they made a change.

www.it-ebooks.info

■ **Note** In SQL Server 2008 the SQL Server Audit feature eliminated the need for many audit trails in triggers, and two other features called change tracking and change data capture may obviate the necessity to use triggers to implement tracking of database changes in triggers. However, using triggers is still a possible tool for building an audit trail depending on your requirements and version/edition of SQL Server. Auditing is covered in Chapter 9 in the "Server and Database Audit" section. I will not cover change tracking and change data capture because they are mostly programming/ETL building tools.

An audit trail is straightforward to implement using triggers. In our example, we'll build an `employee` table and audit any change to the table. I'll keep it simple and have a copy of the table that has a few extra columns for the date and time of the change, plus the user who made the change and what the change was. In Chapter 12, I will talk a bit more about audit trail type objects with an error log. This particular type of audit trail I am discussing in this chapter is typically there for an end user's needs. In this case, the purpose of the `auditTrail` is not necessarily for a security purpose but more to allow the human resources managers to see changes in the employee's status.

We implement an `employee` table (using names with underscores just to add variety) and then a replica to store changes into:

```
CREATE SCHEMA hr;
GO
CREATE TABLE hr.employee
(
  employee_id char(6) NOT NULL CONSTRAINT PKhr_employee PRIMARY KEY,
  first_name varchar(20) NOT NULL,
  last_name varchar(20) NOT NULL,
  salary    decimal(12,2) NOT NULL
);

CREATE TABLE hr.employee_auditTrail
(
  employee_id char(6) NOT NULL,
  date_changed datetime2(0) NOT NULL --default so we don't have to
                                     --code for it
    CONSTRAINT DfltHr_employee_date_changed DEFAULT (SYSDATETIME()),
  first_name varchar(20) NOT NULL,
  last_name varchar(20) NOT NULL,
  salary    decimal(12,2) NOT NULL,
  --the following are the added columns to the original
  --structure of hr.employee
  action char(6) NOT NULL
      CONSTRAINT chkHr_employee_action --we don't log inserts, only changes
      CHECK(action IN ('delete','update')),
  changed_by_user_name sysname NOT NULL
      CONSTRAINT DfltHr_employee_changed_by_user_name
          DEFAULT (ORIGINAL_LOGIN()),
  CONSTRAINT PKemployee_auditTrail PRIMARY KEY (employee_id, date_changed)
);
```


www.it-ebooks.info

Now, we create a trigger with code to determine whether it's an UPDATE or a DELETE, based on how many rows are in the inserted table:

```
CREATE TRIGGER hr.employee$updateAndDeleteAuditTrailTrigger
ON hr.employee
AFTER UPDATE, DELETE AS

BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
 -- @rowsAffected int = (SELECT COUNT(*) FROM inserted);
    @rowsAffected int = (SELECT COUNT(*) FROM deleted);

 BEGIN TRY

      --[validation section]
      --[modification section]

      --since we are only doing update and delete, we just
      --need to see if there are any rows
      --inserted to determine what action is being done.
      DECLARE @action char(6);
      SET @action = CASE WHEN (SELECT COUNT(*) FROM INSERTED) > 0
          THEN 'update' ELSE 'delete' END;

      --since the deleted table contains all changes, we just insert all
      --of the rows in the deleted table and we are done.
      INSERT employee_auditTrail (employee_id, first_name, last_name,
          salary, action)
      SELECT employee_id, first_name, last_name, salary, @action
      FROM deleted;

 END TRY
 BEGIN CATCH
  IF @@trancount > 0
    ROLLBACK TRANSACTION;

    THROW; --will halt the batch or be caught by the caller's catch block
 END CATCH
END;
```

We create some data:

```
INSERT hr.employee (employee_id, first_name, last_name, salary)
VALUES (1, 'Phillip','Taibul',10000);
```


www.it-ebooks.info

Now, unlike the real world in which we live, the person gets a raise immediately:

```
UPDATE hr.employee
SET salary = salary * 1.10 --ten percent raise!
WHERE employee_id = 1;

SELECT *
FROM hr.employee;
```

This returns the data with the new values:

```
employee_id    first_name      last_name     salary
-----------    ----------      ----------    --------
1              Phillip         Taibul        11000.00
```

Check the audit trail table:

```
SELECT *
FROM   hr.employee_auditTrail;
```

You can see that the previous values for the row are stored here:

```
employee_id       date_changed              first_name       last_name
-----------       --------------------      ---------        ---------
1                 2011-06-13 20:18:16       Phillip          Taibul

salary       action      changed_by_user_name
------       ------      --------------------
10000.00     update      DENALI-PC\AlienDrsql
```

This can be a cheap and effective auditing system for many smaller systems. If you have a lot of columns, it can be better to check and see which columns have changed and implement a table that has `tablename`, `columnname`, and previous value columns, but often, this simple strategy works quite well when the volume is low and the number of tables to audit isn't large. Keeping only recent history in the audit trail table helps as well.

Relationships That Span Databases and Servers

Prior to constraints, all relationships were enforced by triggers. Thankfully, when it comes to relationships, triggers are now relegated to enforcing special cases of relationships, such as when you have relationships between tables that are on different databases. I have used this sort of thing when I had a common demographics database that many different systems used.

To implement a relationship using triggers, you need several triggers:

- Parent:
  - `UPDATE`: Disallow the changing of keys if child values exist, or cascade the update.
  - `DELETE`: Prevent or cascade the deletion of rows that have associated parent rows.

www.it-ebooks.info

- Child:
  - INSERT: Check to make sure the key exists in the parent table.
  - UPDATE: Check to make sure the "possibly" changed key exists in the parent table.

To begin this section, I will present templates to use to build these triggers, and then in the final section, I will code a complete trigger for demonstration. For these snippets of code, I refer to the tables as *parent* and *child*, with no schema or database named. Replacing the bits that are inside these greater-than and less-than symbols with appropriate code and table names that include the database and schema gives you the desired result when plugged into the trigger templates we've been using throughout this -chapter.

Parent Update

Note that you can omit the parent update step if using surrogate keys based on identity property columns, because they aren't editable and hence cannot be changed.

There are a few possibilities you might want to implement:

- Cascading operations to child rows
- Preventing updating parent if child rows exist

Cascading operations is not possible from a proper generic trigger coding standpoint. The problem is that if you modify the key of one or more parent rows in a statement that fires the trigger, there is not necessarily any way to correlate rows in the inserted table with the rows in the deleted table, leaving you unable to know which row in the inserted table is supposed to match which row in the deleted table. So, I would not implement the cascading of a parent key change in a trigger; I would do this in your external code (though, frankly, I rarely see the need for modifiable keys anyhow).

Preventing an update of parent rows where child rows exist is very straightforward. The idea here is that you want to take the same restrictive action as the NO ACTION clause on a relationship, for example:

```
IF UPDATE(<parent_key_columns>)

  BEGIN
      IF EXISTS ( SELECT *
                  FROM deleted
                    JOIN <child>
                      ON <child>.<parent_keys> =
                        deleted.<parent_keys>
        )

    BEGIN
      IF @rowsAffected = 1
          SELECT @msg = 'one row message' + inserted.<somedata>
          FROM inserted;
        ELSE
          SELECT @msg = 'multi-row message';
      THROW 50000, @msg, 16;
    END
  END
```


www.it-ebooks.info

Parent Delete

Like the update possibilities, when a parent table row is deleted, we can either:

- Cascade the delete to child rows
- Prevent deleting parent rows if child rows exist

Cascading is very simple. For the delete, you simply use a correlated EXISTS subquery to get matching rows in the child table to the parent table:

```
DELETE <child>
WHERE EXISTS ( SELECT *
               FROM <parent>
               WHERE <child>.<parent_key> = <parent>.<parent_key>);
```

To prevent the delete from happening when a child row exists, here's the basis of code to prevent deleting rows that have associated parent rows:

```
IF EXISTS ( SELECT *
            FROM deleted
              JOIN <child>
                ON <child>.<parent_key> = deleted.<parent_key>
                  )
     BEGIN
       IF @rowsAffected = 1
         SELECT @msg = 'one row message' + inserted.<somedata>
         FROM inserted;
       ELSE
         SELECT @msg = 'multi-row message'
       THROW 50000, @msg, 16;
     END
END
```

Child Insert and Child Update

On the child table, the goal will basically be to make sure that for every value you create in the child table, there exists a corresponding row in the parent table. The following snippet does this and takes into consideration the case where null values are allowed as well:

```
 --@numrows is part of the standard template
 DECLARE @nullcount int,
         @validcount int;

 IF UPDATE(<parent_key>)
  BEGIN
    --you can omit this check if nulls are not allowed
    SELECT   @nullcount = COUNT(*)
    FROM     inserted
    WHERE    inserted.<parent_key> IS NULL;

    --does not count null values
    SELECT @validcount = COUNT(*)
```


www.it-ebooks.info

```
    FROM inserted
          JOIN <parent> AS Parent
              ON inserted.<parent_keys> = Parent.<parent_keys>;

    if @validcount + @nullcount != @numrows
      BEGIN
        IF @rowsAffected = 1
          SELECT @msg = 'The inserted <parent_key_name>:'
                        + cast(parent_key as varchar(10))
                        + ' is not valid in the parent table.'
          FROM inserted;
        ELSE
          SELECT @msg = 'Invalid <parent key> in the inserted rows;'
        THROW 50000, @msg, 16;
      END
END
```

Using basic blocks of code such as these, you can validate most any foreign key relationship using triggers. For example, say you have a table in your PhoneData database called Logs.Call, with a primary key of CallId. In the CRM database, you have a Contacts.Journal table that stores contacts made to a person. To implement the child update and insert a trigger, just fill in the blanks. (I've put the parts of the code in bold where I've replaced the tags with the text specific to this trigger.)

```
CREATE TRIGGER Contacts.Journal$afterInsertUpdateTrigger
ON Contacts.Journal
AFTER INSERT, UPDATE AS
BEGIN
 SET NOCOUNT ON;
 SET ROWCOUNT 0; --in case the client has modified the rowcount
 --use inserted for insert or update trigger, deleted for update or delete trigger
 --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
 --that is equal to number of merged rows, not rows being checked in trigger
 DECLARE @msg varchar(2000), --used to hold the error message
 --use inserted for insert or update trigger, deleted for update or delete trigger
 --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
 --that is equal to number of merged rows, not rows being checked in trigger
    @rowsAffected int = (SELECT COUNT(*) FROM inserted);
 -- @rowsAffected int = (SELECT COUNT(*) FROM deleted);

 BEGIN TRY
   --[validation section]

   --@numrows is part of the standard template
   DECLARE @nullcount int,
           @validcount int;

   IF UPDATE(CallId)
   BEGIN
     --omit this check if nulls are not allowed
     --(left in here for an example)
     SELECT @nullcount = COUNT(*)
     FROM inserted
```


www.it-ebooks.info

```
   WHERE inserted.CallId IS NULL;

   --does not include null values
   SELECT @validcount = COUNT(*)
   FROM inserted
       JOIN PhoneData.Logs.Call AS Parent
         ON inserted.CallId = Parent.CallId;

   IF @validcount + @nullcount <> @numrows
     BEGIN
       IF @rowsAffected = 1
          SELECT @msg = 'The inserted CallId: '
                + cast(CallId AS varchar(10))
                + ' is not valid in the'
                + ' PhoneData.Logs.Call table.'
          FROM inserted;
       ELSE
          SELECT @msg = 'Invalid CallId in the inserted rows.';

       THROW 50000, @ErrorMessage, @ErrorState;

    END
 END

      --[modification section]

 END TRY
 BEGIN CATCH
         IF @@trancount > 0
               ROLLBACK TRANSACTION
         THROW; --will halt the batch or be caught by the caller's catch block

 END CATCH
END
```

INSTEAD OF Triggers

As explained in the introduction to the DML triggers section, `INSTEAD OF` triggers fire before to the DML action being affected by the SQL engine, rather than after it for `AFTER` triggers. In fact, when you have an `INSTEAD OF` trigger on a table, it's the first thing that's done when you `INSERT`, `UPDATE`, or `DELETE` from a table. These triggers are named `INSTEAD OF` because they fire *instead of* the native action the user executed. Inside the trigger, you perform the action—either the action that the user performed or some other action. One thing that makes these triggers useful is that you can use them on views to make noneditable views editable. Doing this, you encapsulate calls to all the affected tables in the trigger, much like you would a stored procedure, except now this view has all the properties of a physical table, hiding the actual implementation from users.

Probably the most obvious limitation of `INSTEAD OF` triggers is that you can have only one for each action (`INSERT`, `UPDATE`, and `DELETE`) on the table. It is also possible to combine triggered actions just like you can for `AFTER` triggers, like having one instead of trigger for `INSERT` and `UPDATE` (something I strongly suggest against for almost all uses `INSTEAD OF` triggers). We'll use a slightly modified version of the same trigger template that we used for the T-SQL `AFTER` triggers, covered in more detail in Appendix B.

I most often use `INSTEAD OF` triggers to set or modify values in my statements automatically so that the values are set to what I want, no matter what the client sends in a statement. A good example is a column to record the last time the row was modified. If you record last update times through client calls, it can be

www.it-ebooks.info

problematic if one of the client's clock is a minute, a day, or even a year off. (You see this all the time in applications. My favorite example was in one system where phone calls appeared to be taking negative amounts of time because the client was reporting when something started and the server was recording when it stopped.) It's generally a best practice not to use INSTEAD OF triggers to do validations and to use them only to shape the way the data is seen by the time it's stored in the DBMS. There's one slight alteration to this, in that you can use INSTEAD OF triggers to prevalidate data so that it's never subject to constraints or AFTER triggers. (We will demonstrate that later in this section.)

I'll demonstrate three ways you can use INSTEAD OF triggers:

- Formatting user input
- Redirecting invalid data to an exception table
- Forcing no action to be performed on a table, even by someone who technically has proper rights

In Chapter 6, we used INSTEAD OF triggers to automatically set the value for columns—in that case, the rowCreate and rowModifyDate—so I won't duplicate that example here in this chapter.

Formatting User Input

Consider the columns firstName and lastName. What if the users who were entering this were heads-down, paid-by-the-keystroke kinds of users? Would we want them to go back and futz around with "joHnson" and make sure that it was formatted as "Johnson"? Or what about data received from services that still use mainframes, in which lowercase letters are still considered a work of the underlord? We don't want to have to make anyone go in and reformat the data by hand (even the newbie intern who doesn't know any better).

One good place for this kind of operation is an INSTEAD OF trigger, often using a function to handle the formatting. Here, I'll present them both in their basic state, generally capitalizing the first letter of each word. This way, we can handle names that have two parts, such as Von Smith, or other more reasonable names that are found in reality. The crux of the function is that I'm simply capitalizing the first character of every letter after a space. The function needs to be updated to handle special cases, such as McDonald.

I am going to simply code this function in T-SQL. The syntax for functions hasn't changed much since SQL Server 2000, though in 2005, Microsoft did get a bit more lenient on what you're allowed to call from a function, such as CURRENT_TIMESTAMP—the standard version of GETDATE()—which was one of the most requested changes to functions in SQL Server 2000.

First we start with the following table and definition:

```
CREATE SCHEMA school;
GO
CREATE TABLE school.student
(
  studentId int identity NOT NULL
    CONSTRAINT PKschool_student PRIMARY KEY,
  studentIdNumber char(8) NOT NULL
    CONSTRAINT AKschool_student_studentIdNumber UNIQUE,
  firstName varchar(20) NOT NULL,
  lastName varchar(20) NOT NULL,
```


www.it-ebooks.info

```
 --implementation columns, we will code for them in the trigger too

  rowCreateDate datetime2(3) NOT NULL
    CONSTRAINT dfltSchool_student_rowCreateDate
        DEFAULT (CURRENT_TIMESTAMP),
  rowCreateUser sysname NOT NULL
    CONSTRAINT dfltSchool_student_rowCreateUser DEFAULT (CURRENT_USER)
);
```

Then, we will create a simple function to format a string value. I will use T-SQL to keep it simple for you to implement and use, but if this were going to be heavily used by many different systems, it would probably behoove you to build a proper function that deals with all of the cases (including allowing exceptions, since some people have nonstandard names!). You might want to use CLR, and you would probably also want versions to use for address standardization, phone number format checking, and so on.

```
CREATE FUNCTION Utility.TitleCase
(
  @inputString varchar(2000)
)
RETURNS varchar(2000) AS

BEGIN
  -- set the whole string to lower
  SET @inputString = LOWER(@inputstring);
  -- then use stuff to replace the first character
  SET @inputString =
  --STUFF in the uppercased character in to the next character,
  --replacing the lowercased letter
  STUFF(@inputString,1,1,UPPER(SUBSTRING(@inputString,1,1)));

  --@i is for the loop counter, initialized to 2
  DECLARE @i int = 2;

  --loop from the second character to the end of the string
  WHILE @i < LEN(@inputString)
  BEGIN
    --if the character is a space
    IF SUBSTRING(@inputString,@i,1) = ' '
    BEGIN
      --STUFF in the uppercased character into the next character
      SET @inputString = STUFF(@inputString,@i +
          1,1,UPPER(SUBSTRING(@inputString,@i + 1,1)));
    END
    --increment the loop counter
    SET @i = @i + 1;
  END
  RETURN @inputString;
END;
```


www.it-ebooks.info

Now, we can alter our trigger from the previous section, which was used to set the rowCreateDate rowCreate user for the school.student table. This time, you'll modify the trigger to title-case the name of the student. The changes are in bold:

```
CREATE TRIGGER school.student$insteadOfInsertTrigger
ON school.student
INSTEAD OF INSERT AS
BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
    @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  -- @rowsAffected int = (SELECT COUNT(*) FROM deleted);

  BEGIN TRY
        --[validation section]
        --[modification section]
        --<perform action>
        INSERT INTO school.student(studentIdNumber, firstName, lastName,
                rowCreateDate, rowCreateUser)
        SELECT studentIdNumber,

        Utility.titleCase(firstName),
        Utility.titleCase(lastName),
                    CURRENT_TIMESTAMP, ORIGINAL_LOGIN()
              FROM inserted; --no matter what the user put in the inserted row
  END TRY                    --when the row was created, these values will be inserted

  BEGIN CATCH
        IF @@trancount > 0
          ROLLBACK TRANSACTION;
        THROW; --will halt the batch or be caught by the caller's catch block
    END CATCH
END;
```

Then, insert a new row with funky formatted data:

```
INSERT school.student(studentIdNumber, firstName, lastName)
VALUES ( '0000001','CaPtain', 'von nuLLY');
```

And include two more rows in a single insert with equally funkily formatted data:

```
INSERT school.student(studentIdNumber, firstName, lastName)
VALUES  ( '0000002','NORM', 'uLl'),
        ( '0000003','gREy', 'tezine');
```


www.it-ebooks.info

Next, we can check the data using a simple SELECT statement:

```
SELECT *
FROM school.student
```

Now, you see that this data has been formatted:

```
studentId     studentIdNumber     firstName
---------     ---------------     --------
1             0000001             Captain
2             0000002             Norm
3             0000003             Grey

lastName      rowCreateDate                rowCreateUser
---------     -----------------------      --------------------
Von Nully     2011-06-13 21:01:39.177      DENALI-PC\AlienDrsql
Ull           2011-06-13 21:01:39.177      DENALI-PC\AlienDrsql
Tezine        2011-06-13 21:01:39.177      DENALI-PC\AlienDrsql
```

I'll leave it to you to modify this trigger for the UPDATE version, because there are few differences, other than updating the row rather than inserting it.

It is not uncommon for this kind of formatting to be done at the client to allow for overriding as needed. Just as I have said many times, T-SQL code could (and possibly **should**) be used to manage formatting when it is **always** done. If there are options to override, then you cannot exclusively use a trigger for sure. In our example, you could just use the INSERT trigger to format the name columns initially and then not have an UPDATE trigger to allow for overrides. Of course this largely depends on the requirements of the system, but in this chapter my goal is to present you with options for how to protect your data integrity.

■ **Tip** If we were to run `SELECT SCOPE_IDENTITY()`, it would return `NULL` (because the actual insert was out of scope). Instead of `SCOPE_IDENTITY()`, use the alternate key, in this case, the `studentIdNumber` that equals `'0000001'`. You might also want to forgo using an `IDENTITY` based value for a surrogate key and use a `SEQUENCE` object to generate surrogate values in the case where another suitable candidate key can be found for that table.

Redirecting Invalid Data to an Exception Table

On some occasions, instead of returning an error when an invalid value is set for a column, you simply want to ignore it and log that an error had occurred. Generally, this wouldn't be used for bulk loading data (using SSIS's facilities to do this is a much better idea), but some examples of why you might do this follow:

- *Heads-down key entry*: In many shops where customer feedback or payments are received by the hundreds or thousands, there are people who open the mail, read it, and key in what's on the page. These people become incredibly skilled in rapid entry and generally make few mistakes. The mistakes they do make don't raise an error on their screens; rather, they fall to other people—exception handlers—to fix. You could use an `INSTEAD OF` trigger to redirect the wrong data to an exception table to be handled later.

www.it-ebooks.info

- *Values that are read in from devices*: An example of this is on an assembly line, where a reading is taken but is so far out of range it couldn't be true, because of the malfunction of a device or just a human moving a sensor. Too many exception rows would require a look at the equipment, but only a few might be normal and acceptable. Another possibility is when someone scans a printed page using a scanner and inserts the data. Often, the values read are not right and have to be checked manually.

For our example, I'll design a table to take weather readings from a single thermometer. Sometimes, this thermometer sends back bad values that are impossible. We need to be able to put in readings, sometimes many at a time, because the device can cache results for some time if there is signal loss, but it tosses off the unlikely rows.

We build the following table, initially using a constraint to implement the simple sanity check. In the analysis of the data, we might find anomalies, but in this process, all we're going to do is look for the "impossible" cases:

```
CREATE SCHEMA Measurements;
GO
CREATE TABLE Measurements.WeatherReading
(
  WeatherReadingId int NOT NULL IDENTITY
    CONSTRAINT PKWeatherReading PRIMARY KEY,
  ReadingTime datetime2(3) NOT NULL
    CONSTRAINT AKMeasurements_WeatherReading_Date UNIQUE,
  Temperature float NOT NULL
    CONSTRAINT chkMeasurements_WeatherReading_Temperature
      CHECK(Temperature between -80 and 150)
      --raised from last edition for global warming
);
```

Then, we go to load the data, simulating what we might do when importing the data all at once:

```
INSERT  into Measurements.WeatherReading (ReadingTime, Temperature)
VALUES  ('20080101 0:00',82.00), ('20080101 0:01',89.22),
        ('20080101 0:02',600.32),('20080101 0:03',88.22),
        ('20080101 0:04',99.01);
```

As we know with CHECK constraints, this isn't going to fly:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint
"chkMeasurements_WeatherReading_Temperature". The conflict occurred in database "Chapter7",
table "Measurements.WeatherReading", column 'Temperature'.
```

Select all the data in the table, and you'll see that this data never gets entered. Does this mean we have to dig through every row individually? Yes, in the current scheme. Or you could insert each row individually, which would take a lot more work for the server, but if you've been following along, you know we're going to write an INSTEAD OF trigger to do this for us. First we add a table to hold the exceptions to the Temperature rule:

```
CREATE TABLE Measurements.WeatherReading_exception
(
  WeatherReadingId int NOT NULL IDENTITY
    CONSTRAINT PKMeasurements_WeatherReading_exception PRIMARY KEY,
  ReadingTime datetime2(3) NOT NULL,
  Temperature float NOT NULL
);
```


www.it-ebooks.info

Then, we create the trigger:

```
CREATE TRIGGER Measurements.WeatherReading$InsteadOfInsertTrigger
ON Measurements.WeatherReading
INSTEAD OF INSERT AS

BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
    @rowsAffected int = (SELECT COUNT(*) FROM inserted);
 -- @rowsAffected int = (SELECT COUNT(*) FROM deleted);

 BEGIN TRY

  --[validation section]
  --[modification section]

  --<perform action>

  --BAD data
  INSERT Measurements.WeatherReading_exception
      (ReadingTime, Temperature)
  SELECT ReadingTime, Temperature
  FROM inserted
  WHERE NOT(Temperature BETWEEN -80 and 120);

  --GOOD data
  INSERT Measurements.WeatherReading (ReadingTime, Temperature)
  SELECT ReadingTime, Temperature
  FROM inserted
  WHERE (Temperature BETWEEN -80 and 120);
 END TRY
 BEGIN CATCH
 IF @@trancount > 0
    ROLLBACK TRANSACTION;

 THROW; --will halt the batch or be caught by the caller's catch block
 END CATCH
END
```

Now, we try to insert the rows with the bad data still in there:

```
INSERT INTO Measurements.WeatherReading (ReadingTime, Temperature)
VALUES  ('20080101 0:00',82.00), ('20080101 0:01',89.22),
        ('20080101 0:02',600.32),('20080101 0:03',88.22),
        ('20080101 0:04',99.01);

SELECT *
FROM Measurements.WeatherReading;
```


www.it-ebooks.info

The good data is in the following output:

```
WeatherReadingId     ReadingTime                Temperature
----------------     -----------------------    -----------
4                    2008-01-01 00:00:00.000    82
5                    2008-01-01 00:01:00.000    89.22
6                    2008-01-01 00:03:00.000    88.22
7                    2008-01-01 00:04:00.000    99.01
```

The nonconforming data can be seen by viewing the data in the exception table:

```
SELECT *
FROM Measurements.WeatherReading_exception;
```

This returns the following result:

```
WeatherReadingId     ReadingTime                Temperature
----------------     -----------                -----------
1                    2008-01-01 00:02:00.000    600.32
```

Now, it might be possible to go back and work on each exception, perhaps extrapolating the value it should have been, based on the previous and the next measurements taken:

```
(88.22 + 89.22) /2 = 88.72
```

Of course, if we did that, we would probably want to include another attribute that indicated that a reading was extrapolated rather than an actual reading from the device. This is obviously a very simplistic example, and you could even make the functionality a lot more interesting by using previous readings to determine what is reasonable.

Forcing No Action to Be Performed on a Table

Our final `INSTEAD OF` trigger example deals with what's almost a security issue. Often, users have *too* much access, and this includes administrators who generally use `sysadmin` privileges to look for problems with systems. Some tables we simply don't ever want to be modified. We might implement triggers to keep any user—even a system administrator—from changing the data.

In this example, we're going to implement a table to hold the version of the database. It's a single-row "table" that behaves more like a global variable. It's here to tell the application which version of the schema to expect, so it can tell the user to upgrade or lose functionality:

```
CREATE SCHEMA System;
go
CREATE TABLE System.Version
(
  DatabaseVersion varchar(10) NOT NULL
);
INSERT into System.Version (DatabaseVersion)
VALUES ('1.0.12');
```


www.it-ebooks.info

Our application always looks to this value to see what objects it expects to be there when it uses them. We clearly don't want this value to get modified, even if someone has db_owner rights in the database. So, we might apply an INSTEAD OF trigger:

```
CREATE TRIGGER System.Version$InsteadOfInsertUpdateDeleteTrigger
ON System.Version
INSTEAD OF INSERT, UPDATE, DELETE AS

BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
     @rowsAffected int = (SELECT COUNT(*) FROM inserted);

  IF @rowsAffected = 0 SET @rowsAffected = (SELECT COUNT(*) FROM deleted);

  --no need to complain if no rows affected

  IF @rowsAffected = 0 RETURN;

  --No error handling necessary, just the message.
  --We just put the kibosh on the action.
  THROW 50000, 'The System.Version table may not be modified in production', 16;
END;
```

Attempts to delete the value, like so

```
UPDATE system.version
SET    DatabaseVersion = '1.1.1';
GO
```

will result in the following:

```
Msg 50000, Level 16, State 16, Procedure Version$InsteadOfInsertUpdateDeleteTrigger, Line 15

The System.Version table may not be modified in production
```

Checking the data, you will see that it remains the same:

```
SELECT *
FROM   System.Version;

Returns:
```

```
DatabaseVersion
---------------
1.0.12
```


www.it-ebooks.info

The administrator, when doing an upgrade, would then have to take the conscious step of running the following code:

```
ALTER TABLE system.version
    DISABLE TRIGGER version$InsteadOfInsertUpdateDeleteTrigger;
```

Now, you can run the update statement:

```
UPDATE system.version
SET    DatabaseVersion = '1.1.1';
```

And checking the data

```
SELECT *
FROM   System.Version;
```

you will see that it has been modified:

```
DatabaseVersion
---------------
1.1.1
```

Reenable the trigger using `ALTER TABLE . . . ENABLE TRIGGER`:

```
ALTER TABLE system.version
    ENABLE TRIGGER version$InsteadOfInsertUpdateDeleteTrigger;
```

Using a trigger like this (not disabled, of course, which is something you can catch with a DDL trigger) enables you to "close the gate," keeping the data safely in the table, even from accidental changes.

Dealing with Triggers and Constraints Errors

One important thing to consider about triggers and constraints is how you need to deal with the error-handling errors caused by constraints or triggers. One of the drawbacks to using triggers is that the state of the database after a trigger error is different from when you have a constraint error. This is further complicated by the changes that were in SQL Server 2005 to support `TRY-CATCH`.

In versions of SQL Server prior to the implementation of `TRY-CATCH`, handling errors for triggers was easy—if there's an error in a trigger, everything stops in its tracks. Now, this has changed. We need to consider two situations when we do a `ROLLBACK` in a trigger, using an error handler such as we have in this chapter:

- **You aren't using a `TRY-CATCH` block**: This situation is simple. The batch stops processing in its tracks. SQL Server handles cleanup for any transaction you were in.
- **You are using a `TRY-CATCH` block**: This situation can be a bit tricky.

Take a `TRY-CATCH` block, such as this one:

```
BEGIN TRY
        <DML STATEMENT>
END TRY
BEGIN CATCH
        <handle it>
END CATCH
```


www.it-ebooks.info

If the T-SQL trigger rolls back and an error is raised, when you get to the <handle it> block, you won't be in a transaction. For CLR triggers, you're in charge of whether the connection ends. When a CHECK constraint causes the error or executes a simple RAISERROR, you'll be in a transaction. Generically, here's the CATCH block that I use (making use of the objects we've already been using in the triggers):

```
    BEGIN CATCH
        IF @@trancount > 0
            ROLLBACK TRANSACTION;
        THROW; --will halt the batch or be caught by the caller's catch block
  END CATCH
```

In almost every case, I roll back any transaction, log the error, and then reraise the error. As an example, I will build the following abstract tables for demonstrating trigger and constraint error handling:

```
CREATE SCHEMA alt;
GO
CREATE TABLE alt.errorHandlingTest
(
    errorHandlingTestId int NOT NULL CONSTRAINT PKerrorHandlingTest PRIMARY KEY,
    CONSTRAINT chkAlt_errorHandlingTest_errorHandlingTestId_greaterThanZero
        CHECK (errorHandlingTestId > 0)
);
GO
```

Note that if you try to put a value greater than 0 into the errorHandlingTestId, it will cause a constraint error. In the trigger, the only statement we will implement in the TRY section will be to raise an error. So no matter what input is sent to the table, it will be discarded and an error will be raised and as we have done previously, we will use ROLLBACK if there is a transaction in progress and then do a THROW.

```
CREATE TRIGGER alt.errorHandlingTest$afterInsertTrigger
ON alt.errorHandlingTest
AFTER INSERT
AS
    BEGIN TRY
      THROW 50000, 'Test Error',16;
    END TRY
    BEGIN CATCH
      IF @@TRANCOUNT > 0
              ROLLBACK TRANSACTION;
      THROW;
    END CATCH
GO
```

The first thing to understand is that when a constraint causes the DML operation to fail, the batch will continue to operate:

```
--NO Transaction, Constraint Error
INSERT alt.errorHandlingTest
VALUES (-1);
SELECT 'continues';
```


www.it-ebooks.info

You will see that the error is raised, and then the SELECT statement is executed:

```
Msg 547, Level 16, State 0, Line 2

The INSERT statement conflicted with the CHECK constraint "chkAlt_errorHandlingTest_
errorHandlingTestId_greaterThanZero". The conflict occurred in database "Chapter7", table
"alt.errorHandlingTest", column 'errorHandlingTestId'.

The statement has been terminated.

---------
continues
```

However, do this with a trigger error:

```
INSERT alt.errorHandlingTest
VALUES (1);
SELECT 'continues';
```

This returns the following and does not get to the SELECT 'continues' line at all:

```
Msg 50000, Level 16, State 16, Procedure errorHandlingTest$afterInsertTrigger, Line 6

Test Error
```

There are also differences in dealing with errors from constraints and triggers when you are using TRY-CATCH and transactions. Take the following batch. The error will be a constraint type. The big thing to understand is the state of a transaction after the error. This is definitely an issue that you have to be careful with.

```
BEGIN TRY
    BEGIN TRANSACTION
    INSERT alt.errorHandlingTest
    VALUES (-1);
    COMMIT
END TRY
BEGIN CATCH
    SELECT CASE XACT_STATE()
            WHEN 1 THEN 'Committable'
            WHEN 0 THEN 'No transaction'
            ELSE 'Uncommitable tran' END as XACT_STATE
        ,ERROR_NUMBER() AS ErrorNumber
        ,ERROR_MESSAGE() as ErrorMessage;
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION;
END CATCH
```

This returns the following:

```
XACT_STATE      ErrorNumber     ErrorMessage
-----------     -----------     ------------------------------------------------------------
Committable     547             The INSERT statement conflicted with the CHECK constraint…
```


www.it-ebooks.info

The transaction is still in force and in a stable state. If you wanted to continue on in the batch doing whatever you need to do it is certainly fine to do so. However, if you end up using any triggers to enforce data integrity, the situation will be different. In the next batch, we will use 1 as the value, so we get a trigger error instead of a constraint one:

```
BEGIN TRANSACTION
    BEGIN TRY
        INSERT alt.errorHandlingTest
        VALUES (1);
        COMMIT TRANSACTION;
    END TRY
BEGIN CATCH
    SELECT CASE XACT_STATE()
              WHEN 1 THEN 'Committable'
              WHEN 0 THEN 'No transaction'
              ELSE 'Uncommitable tran' END as XACT_STATE
          ,ERROR_NUMBER() AS ErrorNumber
          ,ERROR_MESSAGE() AS ErrorMessage;
    IF @@TRANCOUNT > 0
          ROLLBACK TRANSACTION;
END CATCH
```

This returns the following:

```
XACT_STATE          ErrorNumber      ErrorMessage
--------------      -----------      ------------
No transaction      50000            Test Error
```

In the error handler of our batch, the session is no longer in a transaction, since we rolled the transaction back in the trigger. However, unlike the case without an error handler, we continue on in the batch rather than the batch dying. Note, however, that there is no way to recover from an issue in a trigger without resorting to trickery (like storing status in a temporary table instead of throwing an error or rolling back, but this is highly discouraged to keep coding standard and easy to follow).

The unpredictability of the transaction state is why we check the `@@TRANCOUNT` to see if we need to do a rollback. In this case, the error message in the trigger was bubbled up into this `CATCH` statement, so we are in an error state that is handled by the `CATCH BLOCK`.

As a final demonstration, let's look at one other case, and that is where you raise an error in a trigger without rolling back the transaction.

```
ALTER TRIGGER alt.errorHandlingTest$afterInsertTrigger
ON alt.errorHandlingTest
AFTER INSERT
```


www.it-ebooks.info

```
AS
        BEGIN TRY
            THROW 50000, 'Test Error',16;
        END TRY
        BEGIN CATCH
          --Commented out for test purposes
          --IF @@TRANCOUNT > 0
          -- ROLLBACK TRANSACTION;
          THROW;
        END CATCH
```

Now, causing an error in the trigger:

```
BEGIN TRY
    BEGIN TRANSACTION
    INSERT alt.errorHandlingTest
    VALUES (1);
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    SELECT CASE XACT_STATE()
                WHEN 1 THEN 'Committable'
                WHEN 0 THEN 'No transaction'
                ELSE 'Uncommitable tran' END as XACT_STATE
              ,ERROR_NUMBER() AS ErrorNumber
              ,ERROR_MESSAGE() as ErrorMessage;
       IF @@TRANCOUNT > 0
          ROLLBACK TRANSACTION;
END CATCH
```

The result will be as follows:

```
XACT_STATE            ErrorNumber      ErrorMessage
----------------      -----------      ------------
Uncommitable tran     50000            Test Error
```

You get an uncommittable transaction, which is also called "doomed" sometimes. A doomed transaction is still in force but can never be committed and must be rolled back.

The point to all of this is that you need to be careful when you code your error handling to do a few things:

- **Keep things simple:** Do only as much handling as you need, and generally treat errors as unrecoverable unless recovery is truly necessary. The key is to deal with the errors and get back out to a steady state so that the client can know what to try again.
- **Keep things standard:** Set a standard, and follow it. Always use the same handler for all your code in all cases where it needs to do the same things.
- **Test well:** The most important bit of information is to test and test again all the possible paths your code can take.

To always get a consistent situation in my code, I pretty much always use a standard handler. Basically, before every data manipulation statement, I set a manual message in a variable, use it as the first half of the

www.it-ebooks.info

message to know what was being executed, and then append the system message to know what went wrong, sometimes using a constraint mapping function as mentioned earlier, although usually that is overkill since the UI traps all errors:

```
BEGIN TRY
    BEGIN TRANSACTION;
    DECLARE @errorMessage nvarchar(4000) = 'Error inserting data into alt.errorHandlingTest';
    INSERT alt.errorHandlingTest
    VALUES (-1);
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION;

    --I also add in the stored procedure or trigger where the error
    --occurred also when in a coded object
    SET @errorMessage = Coalesce(@errorMessage,'') +
          ' ( System Error: ' + CAST(ERROR_NUMBER() AS varchar(10)) +
          ':' + ERROR_MESSAGE() + ': Line Number:' +
         CAST(ERROR_LINE() AS varchar(10)) + ')';
        THROW 50000,@errorMessage,16;
END CATCH
```

Now, this returns the following:

```
Msg 50000, Level 16, State 16, Line 18
Error inserting data into alt.errorHandlingTest ( System Error: 547:The INSERT statement
conflicted with the CHECK constraint "chkAlt_errorHandlingTest_errorHandlingTestId_
greaterThanZero".
The conflict occurred in database "Chapter7", table "alt.errorHandlingTest", column
'errorHandlingTestId'.: Line Number:4)
```

This returns the manually created message and the system message, as well as where the error occurred. I might also include the call to the `utility.ErrorLog$insert` object, depending on whether the error was something that you expected to occur on occasion or whether it is something (as I said about triggers) that really shouldn't happen. If I was implementing the code in such a way that I expected errors to occur, I might also include a call to something like the `utility.ErrorMap$ MapError` procedure that was discussed earlier to beautify the error message value for the system error.

Error handling did take a leap of improvement in 2005, with a few improvements here in 2011 (most notably being able to rethrow an error using `THROW`, which we have used in the standard trigger template), but it is still not perfect or straightforward. As always, the most important part of writing error handing code is the testing you do to make sure that it works!

Best Practices

The main best practice is to use the right tool for the job. There are many tools in (and around) SQL to use to protect the data. Picking the right tool for a given situation is essential. For example, every column in every table could be defined as `nvarchar(max)`. Using `CHECK` constraints, you could then constrain the values to look like

www.it-ebooks.info

almost any datatype. It sounds silly perhaps, but it is possible. But you know better after reading Chapter 5 and now this chapter, right?

When choosing your method of protecting data, it's best to apply the following types of objects, in this order:

- *Datatypes*: Choosing the right type is the first line of defense. If all your values need to be integers between 1 and 10,000, just using an `integer` datatype takes care of one part of the rule immediately.
- *Defaults*: Though you might not think defaults can be considered data-protection resources, you should know that you can use them to automatically set columns where the purpose of the column might not be apparent to the user (and the database adds a suitable value for the column).
- *Simple* `CHECK` *constraints*: These are important in ensuring that your data is within specifications. You can use almost any scalar functions (user-defined or system), as long as you end up with a single logical expression.
- *Complex* `CHECK` *constraints using functions*: These can be very interesting parts of a design but should be used sparingly, and you should rarely use a function that references the same tables data due to inconsistent results.
- *Triggers*: These are used to enforce rules that are too complex for `CHECK` constraints. Triggers allow you to build pieces of code that fire automatically on any `INSERT`, `UPDATE`, and `DELETE` operation that's executed against a single table.

Don't be afraid to enforce rules in more than one location. Although having rules as close to the data storage as possible is essential to trusting the integrity of the data when you use the data, there's no reason why the user needs to suffer through a poor user interface with a bunch of simple text boxes with no validation. If the tables are designed and implemented properly, you *could* do it this way, but the user should get a nice rich interface as well.

Or course, not all data protection can be done at the object level, and some will need to be managed using client code. This is important for enforcing rules that are optional or frequently changing. The major difference between user code and the methods we have discussed so far in the book is that SQL Server–based enforced integrity is automatic and cannot (accidentally) be overridden. On the other hand, rules implemented using stored procedures or .NET objects cannot be considered as required rules. A simple `UPDATE` statement can be executed from Management Studio that violates rules enforced in a stored procedure.

Summary

Now, you've finished the task of developing the data storage for your databases. If you've planned out your data storage, the only bad data that can get into your system has nothing to do with the design (if a user wants to type the name John as "Jahn" or even "Bill"—stranger things have happened!—there's nothing that can be done in the database server to prevent it). As an architect or programmer, you can't possibly stop users from putting the names of pieces of equipment in a table named `Employee`. There's no semantic checking built in, and it would be impossible to do so without tremendous work and tremendous computing power. Only education can take care of this. Of course, it helps if you've given the users tables to store all their data, but still, users will be users.

The most we can do in SQL Server is to make sure that data is fundamentally sound, such that the data minimally makes sense without knowledge of decisions that were made by the users that, regardless of whether they are correct, are legal values. If your HR employees keep trying to pay your new programmers minimum wage, the database likely won't care, but if they try to say that new employees make a negative salary, actually

www.it-ebooks.info

owing the company money for the privilege to come to work, well, that is probably not going to fly, even if the job is video game tester or some other highly desirable occupation. During this process, we used the resources that SQL Server gives you to protect your data from having such invalid values that would have to be checked for again later.

Once you've built and implemented a set of appropriate data-safeguarding resources, you can then trust that the data in your database has been validated. You should never need to revalidate keys or values in your data once it's stored in your database, but it's a good idea to do random sampling, so you know that no integrity gaps have slipped by you, especially during the full testing process.

www.it-ebooks.info

CHAPTER 8

Patterns and Anti-Patterns

I like rhyme because it is memorable; I like form because having to work to a pattern gives me original ideas.

—Anne Stevenson

There is an old saying that you shouldn't try to reinvent the wheel, and honestly, in essence it is a very good saying. But with all such sayings, a modicum of common sense is required for its application. If everyone down through history took the saying literally, your car would have wheels made out of the trunk of a tree (which the Mythbusters proved you could do in their "Good Wood" episode), since that clearly could have been one of the first wheel-like machines that was used. If everyone down through history had said "that's good enough," driving to Wally World in the family truckster would be a far less comfortable experience.

Over time, however, the basic concept of a wheel has been intact, from rock wheel, to wagon wheel, to steel-belted radials, and even a wheel of cheddar. Each of these is round, able to move itself, and other stuff, by rolling from place A to place B. Each solution follows that common pattern but diverges to solve a particular problem. The goal of a software programmer should be to first try understanding existing techniques and then either use or improve them. Solving the same problem over and over without any knowledge of the past is nuts.

Of course, in as much as there are positive patterns that work, there are also negative patterns that have failed over and over down through history. Take personal flight. For many, many years, truly intelligent people tried over and over to strap wings on their arms or backs and fly. They were close in concept, but just doing the same thing over and over was truly folly. Once it was understood how to apply Bernoulli's principle to building wings and what it would truly take to fly, the Wright Brothers applied this principal, plus principles of lift, to produce the first flying machine. If you ever happen by Kitty Hawk, NC, you can see the plane and location of that flight. Not an amazing amount has changed between that airplane and today's airplanes in basic principle. Once they got it right, it worked.

In designing and implementing a database, you get the very same sort of things going on. The problem with patterns and anti-patterns is that you don't want to squash new ideas immediately. The anti-patterns I will present later in this chapter may be very close to something that becomes a great pattern. Each pattern is there to solve a problem, and in some cases, the problem solved isn't worth the side effects.

Throughout this book so far, we have covered the basic implementation tools that you can use to assemble solutions that meet your real-world needs. In this chapter, I am going to extend this notion and present a few deeper examples where we assemble a part of a database that deals with common problems that show up in almost any database solution. The chapter will be broken up into two major sections. In the first section, we will cover patterns that are common and generally desirable to use. The second half will be anti-patterns, or patterns that you may frequently see that are not desirable to use (along with the preferred method of solution, naturally).

www.it-ebooks.info

Desirable Patterns

In this section, I am going to cover a good variety of implementation patterns that can be used to solve a number of very common problems that you will frequently encounter. By no means should this be confused with a comprehensive list of the types of problems you may face; think of it instead as a sampling of methods of solving some common problems.

The patterns and solutions that I will present are as follows:

- *Uniqueness*: Moving beyond the simple uniqueness we covered in the first chapters of this book, we'll look at some very realistic patterns of solutions that cannot be implemented with a simple uniqueness constraint.
- *Data-driven design*: The goal of data driven design is that you never hard-code values that don't have a fixed meaning. You break down your programming needs into situations that can be based on sets of data values that can be modified without affecting code.
- *Hierarchies*: A very common need is to implement hierarchies in your data. The most common example is the manager-employee relationship. In this section, I will demonstrate the two simplest methods of implementation and introduce other methods that you can explore.
- *Images, documents, and other files*: There is, quite often, a need to store documents in the database, like a web users' avatar picture, or a security photo to identify an employee, or even documents of many types. We will look at some of the methods available to you in SQL Server and discuss the reasons you might choose one method or another.
- *Generalization*: In this section, we will look at some ways that you will need to be careful with how specific you make your tables so that you fit the solution to the needs of the user.
- *Storing user-specified data*: You can't always design a database to cover every known future need. In this section, I will cover some of the possibilities for letting users extend their database themselves in a manner that can be somewhat controlled by the administrators.

■ **Note** I am always looking for other patterns that can solve common issues and enhance your designs (as well as mine). On my web site (`drsql.org`), I may make additional entries available over time, and please leave me comments if you have ideas for more.

Uniqueness

If you have been reading this book straight through, you're probably getting a bit sick of hearing about uniqueness. The fact is, uniqueness is one of the largest problems you will tackle when designing a database, because telling two rows apart from one another can be a very difficult task. Most of our efforts so far have been in trying to tell two rows apart, and that is still a very important task that you always need to do.

But, in this section, we will explore a few more types of uniqueness that hit at the heart of the problems you will come across:

- *Selective*: Sometimes, we won't have all of the information for all rows, but the rows where we do have data need to be unique. As an example, consider the driver's license numbers of employees. No two people can have the same information, but not everyone will necessarily have one.

www.it-ebooks.info

- *Bulk*: Sometimes, we need to inventory items where some of the items are equivalent. For example, cans of corn in the grocery store. You can't tell each item apart, but you do need to know how many you have.
- *Range*: In this case, we want to make sure that ranges of data don't overlap, like appointments. For example, take a hair salon. You don't want Mrs. McGillicutty to have an appointment at the same time as Mrs. Mertz, or no one is going to end up happy.
- *Approximate*: The most difficult case is the most common, in that it can be really difficult to tell two people apart who come to your company for service. Did two Louis Davidsons purchased toy airplanes yesterday? Possibly at the same phone number and address? Probably not, though you can't be completely sure without asking.

Uniqueness is one of the biggest struggles in day-to-day operations, particularly in running a company, as it is sometimes difficult to get customers to divulge identifying information, particularly when they're just browsing. But it is the most important challenge to identify and coalesce unique information so we don't end up with the ten employees with the same SSN numbers, far fewer cans of corn than we expected, ten appointments at the same time, or so we don't send out 12 flyers to the same customer because we didn't get that person uniquely identified.

Selective Uniqueness

We previously discussed `PRIMARY KEY` and `UNIQUE` constraints, but in some situations, neither of these will exactly fit the situation. For example, you may need to make sure some subset of the data, rather than every row, is unique. An example of this is a one-to-one relationship where you need to allow nulls, for example, a `customerSettings` table that lets you add a row for optional settings for a customer. If a user has settings, a row is created, but you want to ensure that only one row is created.

For example, say you have an employee table, and each employee can possibly have an insurance policy. The policy numbers must be unique, but the user might not have a policy.

There are two solutions to this problem that are common:

- *Filtered indexes*: This feature that was new in SQL Server 2008. The `CREATE INDEX` command syntax has a `WHERE` clause so that the index pertains only to certain rows in the table.
- *Indexed view*: In recent versions prior to 2008, the way to implement this is to create a view that has a `WHERE` clause and then index the view.

As a demonstration, I will create a schema and table for the human resources employee table with a column for employee number and insurance policy number as well (the examples in this chapter will be placed in a file named `Chapter8` in the downloads and hence, any error messages will appear there as well).

```
CREATE SCHEMA HumanResources;
GO
CREATE TABLE HumanResources.employee
(
    EmployeeId int NOT NULL IDENTITY(1,1) CONSTRAINT PKalt_employee PRIMARY KEY,
    EmployeeNumber char(5) NOT NULL
        CONSTRAINT AKalt_employee_employeeNummer UNIQUE,
    --skipping other columns you would likely have
    InsurancePolicyNumber char(10) NULL
);
```


www.it-ebooks.info

One of the lesser known but pretty interesting features of indexes is the filtered index. Everything about the index is the same, save for the WHERE clause. So, you add an index like this:

```
--Filtered Alternate Key (AKF)
CREATE UNIQUE INDEX AKFHumanResources_Employee_InsurancePolicyNumber ON

HumanResources.employee(InsurancePolicyNumber)

    WHERE InsurancePolicyNumber IS NOT NULL;
```

Then, create an initial sample row:

```
INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES ('A0001','1111111111');
```

If you attempt to give another employee the same insurancePolicyNumber

```
INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES ('A0002','1111111111');
```

this fails:

```
Msg 2601, Level 14, State 1, Line 1
Cannot insert duplicate key row in object 'HumanResources.employee' with unique index
'AKFHumanResources_Employee_InsurancePolicyNumber'. The duplicate key value is (1111111111).
```

However, adding two rows with null will work fine:

```
INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES  ('A0003','2222222222'),
        ('A0004',NULL),
        ('A0005',NULL);
```

You can see that this:

```
SELECT  *
FROM    HumanResources.Employee;
```

returns the following:

```
EmployeeId      EmployeeNumber  InsurancePolicyNumber
----------      --------------  ---------------------
1               A0001           1111111111
3               A0003           2222222222
4               A0004           NULL
5               A0005           NULL
```

The NULL example is the classic example, because it is common to desire this functionality. However, this technique can be used for more than just NULL exclusion. As another example, consider the case where you want to ensure that only a single row is set as primary for a group of rows, such as a primary contact for an account:

```
CREATE SCHEMA Account;
GO
```


www.it-ebooks.info

```
CREATE TABLE Account.Contact
(
     ContactId varchar(10) NOT NULL,
     AccountNumber char(5) NOT NULL, --would be FK in full example
     PrimaryContactFlag bit NOT NULL,
     CONSTRAINT PKalt_accountContact
         PRIMARY KEY(ContactId, AccountNumber)
);
```

Again, create an index, but this time, choose only those rows with `primaryContactFlag = 1`. The other values in the table could have as many other values as you want (of course, in this case, since it is a bit, the values could be only 0 or 1):

```
CREATE UNIQUE INDEX
     AKFAccount_Contact_PrimaryContact
          ON Account.Contact(AccountNumber)
          WHERE PrimaryContactFlag = 1;
```

If you try to insert two rows that are primary, as in the following statements that will set both contacts `'fred'` and `'bob'` as the primary contact for the account with account number `'11111'`:

```
INSERT INTO Account.Contact
SELECT 'bob','11111',1;
GO
INSERT INTO Account.Contact
SELECT 'fred','11111',1;
```

the following error is returned:

```
Msg 2601, Level 14, State 1, Line 1
Cannot insert duplicate key row in object 'Account.Contact' with unique index
'AKFAccount_Contact_PrimaryContact'. The duplicate key value is (11111).
```

To insert the row with `'fred'` as the name and set it as primary (assuming the `'bob'` row was inserted previously), you will need to update the other row to be not primary and then insert the new primary row:

```
BEGIN TRANSACTION;

UPDATE Account.Contact
SET primaryContactFlag = 0
WHERE accountNumber = '11111';

INSERT Account.Contact
SELECT 'fred','11111', 1;

COMMIT TRANSACTION;
```

Note that in cases like this you would definitely want to use a transaction in your code so you don't end up without a primary contact if the insert fails for some other reason.

Prior to SQL Server 2008, where there were no filtered indexes, the preferred method of implementing this was to create an indexed view. There are a couple of other ways to do this (such as in a trigger or stored procedure using an `EXISTS` query, or even using a user-defined function in a `CHECK` constraint), but the indexed view is the easiest. Then when the insert does its cascade operation to the indexed view, if there are duplicate values, the operation will fail. You can use indexed views in all versions of SQL Server, though only Enterprise

www.it-ebooks.info

Edition will make special use of the indexes for performance purposes. (In other versions, you have to specifically reference the indexed view to realize performance gains. Using indexed views for performance reasons will be demonstrated in Chapter 10.)

Returning to the `InsurancePolicyNumber` uniqueness example, you can create a view that returns all rows other than null `insurancePolicyNumber` values. Note that it has to be schema bound to allow for indexing:

```
CREATE VIEW HumanResources.Employee_InsurancePolicyNumberUniqueness
WITH SCHEMABINDING
AS
  SELECT    InsurancePolicyNumber
  FROM      HumanResources.Employee
  WHERE     InsurancePolicyNumber IS NOT NULL;
```

Now, you can index the view by creating a unique, clustered index on the view:

```
CREATE UNIQUE CLUSTERED INDEX
    AKHumanResources_Employee_InsurancePolicyNumberUniqueness
    ON HumanResources.Employee_InsurancePolicyNumberUniqueness(InsurancePolicyNumber);
```

Now, attempts to insert duplicate values will be met with the following (assuming you drop the existing filtered index, which will be included in the code download.)

```
Msg 2601, Level 14, State 1, Line 1
Cannot insert duplicate key row in object 'HumanResources.
Employee_InsurancePolicyNumberUniqueness' with unique index
'AKHumanResources_Employee_InsurancePolicyNumberUniqueness'.
The duplicate key value is (1111111111).
The statement has been terminated.
```

Both of these techniques are really quite fast and easy to implement. However, the filtered index has a greater chance of being useful for searches against the table, so it is really just a question of education for the programming staff members who might come up against the slightly confusing error messages in their UI or SSIS packages, for example (even with good naming I find I frequently have to look up what the constraint actually does when it makes one of my SSIS packages fail). Pretty much no constraint error should be bubbled up to the end users, unless they are a very advanced group of users, so the UI should be smart enough to either prevent the error from occurring or at least translate it into words that the end user can understand.

Bulk Uniqueness

Sometimes, we need to inventory items where some of the items are equivalent, for example, cans of corn in the grocery store. You can't even tell them apart by looking at them (unless they have different expiration dates, perhaps), but it is a very common need to know how many you have. Implementing a solution that has a row for every canned good in a corner market would require a very large database even for a very small store, and as you sold each item, you would have to allocate those rows as they were sold. This would be really quite complicated and would require a heck of a lot of rows and data manipulation. It would, in fact, make some queries easier, but it would make data storage a lot more difficult.

Instead of having one row for each individual item, you can implement a row per type of item. This type would be used to store inventory and utilization, which would then be balanced against one another. In Figure 8-1, I show a very simplified model of such activity:

In the `InventoryAdjustment` table, you would record shipments coming in, items stolen, changes to inventory after taking inventory (could be more or less, depending on the quality of the data you had), and in the

www.it-ebooks.info

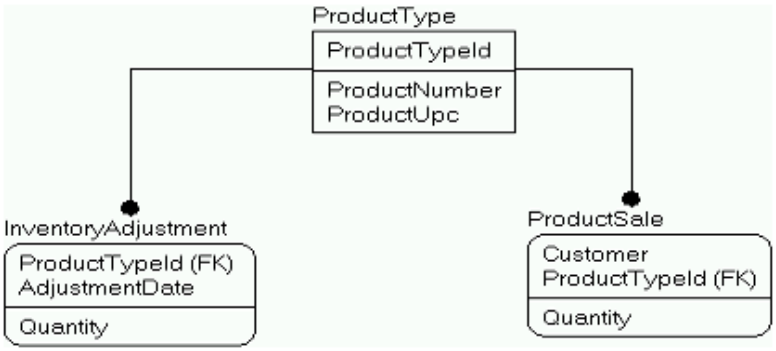


Figure 8-1. *Simplified Inventory Model*

product sale table (probably a sales or invoicing table in a complete model), you record when product is removed from inventory for a good reason.

The sum of the `InventoryAdjustment` `Quantity` less the `ProductSale` `Quantity` should be 0 or greater and should tell you the amount of product on hand. In the more realistic case, you would have a lot of complexity for backorders, future orders, returns, and so on, but the concept is basically the same. Instead of each row representing a single item, it represents a handful of items.

The following miniature design is an example I charge students with when I give my day-long seminar on database design. It is referencing a collection of toys, many of which are exactly alike:

> *A certain person was obsessed with his Lego® collection. He had thousands of them and wanted to catalog his Legos both in storage and in creations where they were currently located and/or used. Legos are either in the storage "pile" or used in a set. Sets can either be purchased, which will be identified by an up to five-digit numeric code, or personal, which have no numeric code. Both styles of set should have a name assigned and a place for descriptive notes.*
> *Legos come in many shapes and sizes, with most measured in 2 or 3 dimensions. First in width and length based on the number of studs on the top, and then sometimes based on a standard height (for example, bricks have height; plates are fixed at 1/3 of 1 brick height unit). Each part comes in many different standard colors as well. Beyond sized pieces, there are many different accessories (some with length/width values), instructions, and so on that can be catalogued.*

Example pieces and sets are shown in Figure 8-2.



Figure 8-2. *Sample Lego® parts for a database*

www.it-ebooks.info

To solve this problem, I will create a table for each type of set of Legos that are owned (which I will call Build, since "set" is a bad word for a SQL name, and "build" actually is better anyhow to encompass a personal creation):

```
CREATE SCHEMA Lego;
GO
CREATE TABLE Lego.Build
(
        BuildId int NOT NULL CONSTRAINT PKLegoBuild PRIMARY KEY,
        Name varchar(30) NOT NULL CONSTRAINT AKLegoBuild_Name UNIQUE,
        LegoCode varchar(5) NULL, --five character set number
        InstructionsURL varchar(255) NULL --where you can get the PDF of the instructions
);
```

Then, add a table for each individual instances of that build, which I will call BuildInstance:

```
CREATE TABLE Lego.BuildInstance
(
        BuildInstanceId Int NOT NULL CONSTRAINT PKLegoBuildInstance PRIMARY KEY ,
        BuildId Int NOT NULL CONSTRAINT FKLegoBuildInstance$isAVersionOf$LegoBuild
             REFERENCES Lego.Build (BuildId),
        BuildInstanceName varchar(30) NOT NULL, --brief description of item
        Notes varchar(1000) NULL, --longform notes. These could describe modifications
             --for the instance of the model
        CONSTRAINT AKLegoBuildInstance UNIQUE(BuildId, BuildInstanceName)
);
```

The next task is to create a table for each individual piece type. I used the term "piece" as a generic version of the different sorts of pieces you can get for Legos, including the different accessories.

```
CREATE TABLE Lego.Piece
(
      PieceId int constraint PKLegoPiece PRIMARY KEY,
      Type  varchar(15) NOT NULL,
      Name  varchar(30) NOT NULL,
      Color varchar(20) NULL,
      Width int NULL,
      Length int NULL,
      Height int NULL,
      LegoInventoryNumber int NULL,
      OwnedCount int NOT NULL,
      CONSTRAINT AKLego_Piece_Definition UNIQUE (Type,Name,Color,Width,Length,Height),
      CONSTRAINT AKLego_Piece_LegoInventoryNumber UNIQUE (LegoInventoryNumber)
);
```

Note that I implement the owned count as an attribute of the piece and not as a multivalued attribute to denote inventory change events. In a fully fleshed out sales model, this might not be sufficient, but for a personal inventory, it would be a reasonable solution. Remember that one of the most important features of a design is to tailor it to the use. The likely use here will be to update the value as new pieces are added to inventory and possibly counting up loose pieces later and adding that value to the ones in sets (which we will have a query for later).

www.it-ebooks.info

Next, I will implement the table to allocate pieces to different builds:

```
CREATE TABLE Lego.BuildInstancePiece
(
    BuildInstanceId int NOT NULL,
    PieceId int NOT NULL,
    AssignedCount int NOT NULL,
    CONSTRAINT PKLegoBuildInstancePiece PRIMARY KEY (BuildInstanceId, PieceId)
);
```

From here, we can load some data. I will load a true item that Lego sells and that I have often given away during presentations. It is a small black one-seat car with a little guy in a sweatshirt.

```
INSERT Lego.Build (BuildId, Name, LegoCode, InstructionsURL)
VALUES (1,'Small Car','3177',
    'http://cache.lego.com/bigdownloads/buildinginstructions/4584500.pdf');
```

I will create one instance for this, as I only personally have one in my collection (plus some boxed ones to give away):

```
INSERT Lego.BuildInstance (BuildInstanceId, BuildId, BuildInstanceName, Notes)
VALUES (1,1,'Small Car for Book', NULL);
```

Then, I load the table with the different pieces in my collection, in this case, the types of pieces included in the set, plus some extras thrown in. (Note that in a fully fleshed out design some of these values would have domains enforced, as well as validations to enforce the types of items that have height, width, and/or lengths. This detail is omitted partially for simplicity, and partially because it might just be too much to implement for a system such as this, based on user needs—though mostly for simplicity of demonstrating the underlying principal of bulk uniqueness in the most compact possible manner.)

```
INSERT Lego.Piece (PieceId, Type, Name, Color, Width, Length, Height,
                   LegoInventoryNumber, OwnedCount)
VALUES (1, 'Brick','Basic Brick','White',1,3,1,'362201',20),
       (2, 'Slope','Slope','White',1,1,1,'4504369',2),
       (3, 'Tile','Groved Tile','White',1,2,NULL,'306901',10),
       (4, 'Plate','Plate','White',2,2,NULL,'302201',20),
       (5, 'Plate','Plate','White',1,4,NULL,'371001',10),
       (6, 'Plate','Plate','White',2,4,NULL,'302001',1),
       (7, 'Bracket','1x2 Bracket with 2x2','White',2,1,2,'4277926',2),
       (8, 'Mudguard','Vehicle Mudguard','White',2,4,NULL,'4289272',1),
       (9, 'Door','Right Door','White',1,3,1,'4537987',1),
       (10,'Door','Left Door','White',1,3,1,'45376377',1),
       (11,'Panel','Panel','White',1,2,1,'486501',1),
       (12,'Minifig Part','Minifig Torso , Sweatshirt','White',NULL,NULL,
           NULL,'4570026',1),
       (13,'Steering Wheel','Steering Wheel','Blue',1,2,NULL,'9566',1),
       (14,'Minifig Part','Minifig Head, Male Brown Eyes','Yellow',NULL, NULL,
           NULL,'4570043',1),
       (15,'Slope','Slope','Black',2,1,2,'4515373',2),
       (16,'Mudguard','Vehicle Mudgard','Black',2,4,NULL,'4195378',1),
       (17,'Tire','Vehicle Tire,Smooth','Black',NULL,NULL,NULL,'4508215',4),
       (18,'Vehicle Base','Vehicle Base','Black',4,7,2,'244126',1),
       (19,'Wedge','Wedge (Vehicle Roof)','Black',1,4,4,'4191191',1),
       (20,'Plate','Plate','Lime Green',1,2,NULL,'302328',4),
       (21,'Minifig Part','Minifig Legs','Lime Green',NULL,NULL,NULL,'74040',1),
```


www.it-ebooks.info

```
        (22,'Round Plate','Round Plate','Clear',1,1,NULL,'3005740',2),
        (23,'Plate','Plate','Transparent Red',1,2,NULL,'4201019',1),
        (24,'Briefcase','Briefcase','Reddish Brown',NULL,NULL,NULL,'4211235', 1),
        (25,'Wheel','Wheel','Light Bluish Gray',NULL,NULL,NULL,'4211765',4),
        (26,'Tile','Grilled Tile','Dark Bluish Gray',1,2,NULL,'4210631', 1),
        (27,'Minifig Part','Brown Minifig Hair','Dark Brown',NULL,NULL,NULL,
            '4535553', 1),
        (28,'Windshield','Windshield','Transparent Black',3,4,1,'4496442',1),
            --and a few extra pieces to make the queries more interesting
        (29,'Baseplate','Baseplate','Green',16,24,NULL,'3334',4),
        (30,'Brick','Basic Brick','White',4,6,NULL,'2356',10);
```

Next, I will assign the 43 pieces that make up the first set (with the most important part of this statement being to show you how cool the row constructor syntax is that was introduced in SQL Server 2008—this would have taken over 20 more lines previously):

```
INSERT INTO Lego.BuildInstancePiece (BuildInstanceId, PieceId, AssignedCount)
VALUES (1,1,2),(1,2,2),(1,3,1),(1,4,2),(1,5,1),(1,6,1),(1,7,2),(1,8,1),(1,9,1),
       (1,10,1),(1,11,1),(1,12,1),(1,13,1),(1,14,1),(1,15,2),(1,16,1),(1,17,4),
       (1,18,1),(1,19,1),(1,20,4),(1,21,1),(1,22,2),(1,23,1),(1,24,1),(1,25,4),
       (1,26,1),(1,27,1),(1,28,1);
```

Finally, I will set up two other minimal builds to make the queries more interesting:

```
INSERT Lego.Build (BuildId, Name, LegoCode, InstructionsURL)
VALUES (2,'Brick Triangle',NULL,NULL);
GO
INSERT Lego.BuildInstance (BuildInstanceId, BuildId, BuildInstanceName, Notes)
VALUES (2,2,'Brick Triangle For Book','Simple build with 3 white bricks');
GO
INSERT INTO Lego.BuildInstancePiece (BuildInstanceId, PieceId, AssignedCount)
VALUES (2,1,3);
GO
INSERT Lego.BuildInstance (BuildInstanceId, BuildId, BuildInstanceName, Notes)
VALUES (3,2,'Brick Triangle For Book2','Simple build with 3 white bricks');
GO
INSERT INTO Lego.BuildInstancePiece (BuildInstanceId, PieceId, AssignedCount)
VALUES (3,1,3);
```

After the mundane business of setting up the scenario is passed, we can count the types of pieces we have in our inventory, and the total number of pieces we have using a query such as this:

```
SELECT COUNT(*) AS PieceCount ,SUM(OwnedCount) AS InventoryCount
FROM Lego.Piece;
```

which returns the following, with the first column giving us the different types.

```
PieceCount   InventoryCount
----------   --------------
30           111
```


www.it-ebooks.info

Here, you start to get a feel for how this is going to be a different sort of solution than the typical SQL solution. Usually, one row represents one thing, but here, you see that, on average, each row represents four different pieces. Following this train of thought, we can group on the generic type of piece using a query such as

```
SELECT Type, COUNT(*) AS TypeCount, SUM(OwnedCount) AS InventoryCount
FROM Lego.Piece
GROUP BY Type;
```

In these results you can see that we have two types of bricks but thirty bricks in inventory, one type of baseplate but four of them, and so on:

```
Type                TypeCount      InventoryCount
---------------     ---------      --------------
Baseplate           1              4
Bracket             1              2
Brick               2              30
Briefcase           1              1
Door                2              2
Minifig Part        4              4
Mudguard            2              2
Panel               1              1
Plate               5              36
Round Plate         1              2
Slope               2              4
Steering Wheel      1              1
Tile                2              11
Tire                1              4
Vehicle Base        1              1
Wedge               1              1
Wheel               1              4
Windshield          1              1
```

The biggest concern with this method is that users have to know the difference between a row and an instance of the thing the row is modeling. And it gets more interesting where the cardinality of the type is very close to the number of physical items on hand. With 30 types of item and only 111 actual pieces, users querying may not immediately see that they are getting a wrong count. In a system with 20 different products and a million pieces of inventory, it will be a lot more obvious.

In the next two queries, I will expand into actual interesting queries that you will likely want to use. First, I will look for pieces that are assigned to a given set, in this case, the small car model that we started with. To do this, we will just join the tables, starting with `Build` and moving on to the `BuildInstance`, `BuildInstancePiece`, and `Piece`. All of these joins are inner joins, since we want items that are included in the set. I use grouping sets (another SQL Server 2008 feature that comes in handy now and again to give us a very specific set of aggregates—in this case, using the `()` notation to give us a total count of all pieces).

```
SELECT CASE WHEN GROUPING(Piece.Type) = 1 THEN '--Total--' ELSE Piece.Type END AS PieceType,
        Piece.Color,Piece.Height, Piece.Width, Piece.Length,
        SUM(BuildInstancePiece.AssignedCount) AS AssignedCount
FROM Lego.Build
        JOIN Lego.BuildInstance
             ON Build.BuildId = BuildInstance.BuildId
```


www.it-ebooks.info

```
        JOIN Lego.BuildInstancePiece
             ON BuildInstance.BuildInstanceId =
                  BuildInstancePiece.BuildInstanceId
        JOIN Lego.Piece
             ON BuildInstancePiece.PieceId = Piece.PieceId
WHERE Build.Name = 'Small Car'
        AND BuildInstanceName = 'Small Car for Book'
GROUP BY GROUPING SETS((Piece.Type,Piece.Color, Piece.Height, Piece.Width, Piece.Length),
             ());
```

This returns the following, where you can see that 43 pieces go into this set:

```
PieceType       Color                Height    Width     Length    AssignedCount
----------      -----------------    ------    -----     ------    -------------
Bracket         White                2         2         1         2
Brick           White                1         1         3         2
Briefcase       Reddish Brown        NULL      NULL      NULL      1
Door            White                1         1         3         2
Minifig         Part Dark Brown      NULL      NULL      NULL      1
Minifig         Part Lime Green      NULL      NULL      NULL      1
Minifig         Part White           NULL      NULL      NULL      1
Minifig         Part Yellow          NULL      NULL      NULL      1
Mudguard        Black                NULL      2         4         1
Mudguard        White                NULL      2         4         1
Panel           White                1         1         2         1
Plate           Lime Green           NULL      1         2         4
Plate           Transparent Red      NULL      1         2         1
Plate           White                NULL      1         4         1
Plate           White                NULL      2         2         2
Plate           White                NULL      2         4         1
Round           Plate Clear          NULL      1         1         2
Slope           Black                2         2         1         2
Slope           White                1         1         1         2
Steering        Wheel Blue           NULL      1         2         1
Tile            Dark Bluish Gray     NULL      1         2         1
Tile            White                NULL      1         2         1
Tire            Black                NULL      NULL      NULL      4
Vehicle         Base Black           2         4         7         1
Wedge           Black                4         1         4         1
Wheel           Light Bluish Gray    NULL      NULL      NULL      4
Windshield      Transparent Black    1         3         4         1
--Total--       NULL                 NULL      NULL      NULL      43
```

The final query in this section is the more interesting one. A very common question would be, how many pieces of a given type do I own that are not assigned to a set? For this, I will use a Common Table Expression (CTE) that gives me a sum of the pieces that have been assigned to a `BuildInstance` and then use that set to join to the `Piece` table:

www.it-ebooks.info

```
;WITH AssignedPieceCount
AS (
SELECT  PieceId, SUM(AssignedCount) AS TotalAssignedCount
FROM    Lego.BuildInstancePiece
GROUP   BY PieceId )

SELECT  Type, Name, Width, Length,Height,
        Piece.OwnedCount - Coalesce(TotalAssignedCount,0) AS AvailableCount
FROM    Lego.Piece
             LEFT OUTER JOIN AssignedPieceCount
                 ON Piece.PieceId = AssignedPieceCount.PieceId
WHERE Piece.OwnedCount - COALESCE(TotalAssignedCount,0) > 0;
```

Because the cardinality of the `AssignedPieceCount` to the `Piece` table is zero or one to one, we can simply do an outer join and subtract the number of pieces we have assigned to sets from the amount owned. This returns

```
Type        Name          Width   Length   Height   AvailableCount
---------   -----------   -----   ------   ------   --------------
Brick       Basic Brick   1       3        1        12
Tile        Groved Tile   1       2        NULL     9
Plate       Plate         2       2        NULL     18
Plate       Plate         1       4        NULL     9
Baseplate   Baseplate     16      24       NULL     4
Brick       Basic Brick   4       6        NULL     10
```

You can expand this basic pattern to most any bulk uniqueness situation you may have. The calculation of how much inventory you have may be more complex and might include inventory values that are stored daily to avoid massive recalculations (think about how your bank account balance is set at the end of the day, and then daily transactions are added/subtracted as they occur until they too are posted and fixed in a daily balance).

Range Uniqueness

In some cases, uniqueness isn't uniqueness on a single column or even a composite set of columns, but rather over a range of values. Very common examples of this include appointment times, college classes, or even teachers/employees who can only be assigned to one location at a time.

We can protect against situations such as overlapping appointment times by employing a trigger and a range overlapping checking query. The toughest part about checking item ranges is that be three basic situations have to be checked. Say you have `appointment1`, and it is defined with precision to the second, and starting on `'20110712 1:00:00PM'`, and ending at `'20110712 1:59:59PM'`. To validate the data, we need to look for rows where any of the following conditions are met, indicating an improper data situation:

- The start or end time for the new appointment falls between the start and end for another appointment
- The start time for the new appointment is before and the end time is after the end time for another appointment

If these two conditions are not met, the new row is acceptable. We will implement a simplistic example of assigning a doctor to an office. Clearly, other parameters that need to be considered, like office space, assistants, and so on, but I don't want this section to be larger than the allotment of pages for the entire book. First, we create a table for the doctor and another to set appointments for the doctor.

www.it-ebooks.info

```
CREATE SCHEMA office;
GO
CREATE TABLE office.doctor
(
        doctorId int NOT NULL CONSTRAINT PKOfficeDoctor PRIMARY KEY,
        doctorNumber char(5) NOT NULL CONSTRAINT AKOfficeDoctor_doctorNumber UNIQUE
);
CREATE TABLE office.appointment
(
        appointmentId int NOT NULL CONSTRAINT PKOfficeAppointment PRIMARY KEY,
        --real situation would include room, patient, etc,
        doctorId   int NOT NULL,
        startTime  datetime2(0) NOT NULL, --precision to the second
        endTime    datetime2(0) NOT NULL,
        CONSTRAINT AKOfficeAppointment_DoctorStartTime UNIQUE (doctorId,startTime),
        CONSTRAINT AKOfficeAppointment_StartBeforeEnd CHECK (startTime <= endTime)
);
```

Next, we will add some data to our new table. The row with `appointmentId` value 5 will include a bad date range that overlaps another row for demonstration purposes:

```
INSERT INTO office.doctor (doctorId, doctorNumber)
VALUES (1,'00001'),(2,'00002');
INSERT INTO office.appointment
VALUES (1,1,'20110712 14:00','20110712 14:59:59'),
        (2,1,'20110712 15:00','20110712 16:59:59'),
        (3,2,'20110712 8:00','20110712 11:59:59'),
        (4,2,'20110712 13:00','20110712 17:59:59'),
        (5,2,'20110712 14:00','20110712 14:59:59'); --offensive item for demo, conflicts
                                                    --with 4
```

Now, we run the following query to test the data:

```
SELECT  appointment.appointmentId,
        Acheck.appointmentId AS conflictingAppointmentId
FROM    office.appointment
        JOIN office.appointment AS ACheck
             ON appointment.doctorId = ACheck.doctorId
        /*1*/ AND appointment.appointmentId <> ACheck.appointmentId
        /*2*/ AND (Appointment.startTime BETWEEN Acheck.startTime and Acheck.endTime
        /*3*/ OR Appointment.endTime BETWEEN Acheck.startTime and Acheck.endTime
        /*4*/ OR (appointment.startTime < Acheck.startTime and appointment.endTime > Acheck.
                 endTime));
```

In this query, I have highlighted four points:

1. In the join, we have to make sure that we don't compare the current row to itself, because an appointment will always overlap itself.
2. Here, we check to see if the `startTime` is between the start and end, inclusive of the actual values.
3. Same as 2 for the `endTime`.
4. Finally, we check to see if any appointment is engulfing another.

www.it-ebooks.info

Running the query, we see that

```
appointmentId   conflictingAppointmentId
-------------   ------------------------
5               4
4               5
```

The interesting part of these results is that you will always get a pair of offending rows, because if one row is offending in one way, like starting before and after another appointment, the conflicting row will have a start and end time between the first appointment's time. This won't a problem, but the shared blame can make the results more interesting to deal with.

Next, we remove the bad row for now:

```
DELETE FROM office.appointment WHERE AppointmentId = 5;
```

We will now implement a trigger (using the template as defined in Appendix B and used in previous chapters) that will check for this condition based on the values in new rows being inserted or updated. There's no need to check the deletion, because all a delete operation can do is help the situation. Note that the basis of this trigger is the query we used previously to check for bad values:

```
CREATE TRIGGER office.appointment$insertAndUpdateTrigger
ON office.appointment
AFTER UPDATE, INSERT AS
BEGIN

  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
        @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  --        @rowsAffected int = (SELECT COUNT(*) FROM deleted);
  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;
  BEGIN TRY
      --[validation section]
    IF UPDATE(startTime) OR UPDATE(endTime) OR UPDATE(doctorId)
      BEGIN
      IF EXISTS ( SELECT *
                  FROM office.appointment
                      JOIN office.appointment AS ACheck
                      on appointment.doctorId = ACheck.doctorId
                   AND appointment.appointmentId <> ACheck.appointmentId
                   AND (Appointment.startTime between Acheck.startTime
                   AND Acheck.endTime
                   OR Appointment.endTime between Acheck.startTime
                   AND Acheck.endTime
```


www.it-ebooks.info

```
                    OR (appointment.startTime < Acheck.startTime
                    AND appointment.endTime > Acheck.endTime))
                   WHERE EXISTS (SELECT *
                     FROM inserted
                     WHERE inserted.doctorId = Acheck.doctorId))
       BEGIN
       IF @rowsAffected = 1
       SELECT @msg = 'Appointment for doctor ' + doctorNumber +
                     'overlapped existing appointment'
       FROM inserted
            JOIN office.doctor
            ON inserted.doctorId = doctor.doctorId;
      ELSE
      SELECT @msg = 'One of the rows caused an overlapping ' +
                    'appointment time for a doctor';
        THROW 50000,@msg,16;
         END
    END
    --[modification section]
    END TRY
    BEGIN CATCH
    IF @@trancount > 0
     ROLLBACK TRANSACTION;
    THROW; --will halt the batch or be caught by the caller's catch block
  END CATCH
END;
GO
```

Next, as a refresher, check out the data that is in the table:

```
SELECT *
FROM office.appointment;
```

This returns (or at least it should, assuming you haven't deleted or added extra data)

```
appointmentId   doctorId   startTime            endTime
-------------   --------   -------------------  -------------------
1               1          2011-07-12 14:00:00  2011-07-12 14:59:59
2               1          2011-07-12 15:00:00  2011-07-12 16:59:59
3               2          2011-07-12 08:00:00  2011-07-12 11:59:59
4               2          2011-07-12 13:00:00  2011-07-12 17:59:59
```

This time, when we try to add an appointment for `doctorId` number 1:

```
INSERT INTO office.appointment
VALUES (5,1,'20110712 14:00','20110712 14:59:59');
```

this first attempt is blocked because the row is an exact duplicate of the start time value. It might seem tricky, but the most common error is often trying to duplicate something accidentally.

www.it-ebooks.info

```
Msg 2627, Level 14, State 1, Line 2
Violation of UNIQUE KEY constraint 'AKOfficeAppointment_DoctorStartTime'. Cannot insert
duplicate key in object 'office.appointment'. The duplicate key value is (1, 2011-07-12
14:00:00).
```

Next, we check the case where the appointment fits wholly inside of another appointment:

```
INSERT INTO office.appointment
VALUES (5,1,'20110712 14:30','20110712 14:40:59');
```

This fails and tells us the doctor for whom the failure occurred:

```
Msg 50000, Level 16, State 16, Procedure appointment$insertAndUpdateTrigger, Line 39
Appointment for doctor 00001 overlapped existing appointment
```

Then, we test for the case where the entire appointment engulfs another appointment:

```
INSERT INTO office.appointment
VALUES (5,1,'20110712 11:30','20110712 14:59:59');
```

This quite obediently fails, just like the other case:

```
Msg 50000, Level 16, State 16, Procedure appointment$insertAndUpdateTrigger, Line 39
Appointment for doctor 00001 overlapped existing appointment
```

And, just to drive home the point of always testing your code extensively, you should always test the greater-than-one-row case, and in this case, I included rows for both doctors (this is starting to sound very Dr. Who–ish):

```
INSERT into office.appointment
VALUES (5,1,'20110712 11:30','20110712 14:59:59'),
       (6,2,'20110713 10:00','20110713 10:59:59');
```

This time, it fails with our multirow error message:

```
Msg 50000, Level 16, State 16, Procedure appointment$insertAndUpdateTrigger, Line 39
One of the rows caused an overlapping appointment time for a doctor
```

Finally, add two rows that are safe to add:

```
INSERT INTO office.appointment
VALUES (5,1,'20110712 10:00','20110712 11:59:59'),
       (6,2,'20110713 10:00','20110713 10:59:59');
```

This will (finally) work. Now, test failing an update operation:

```
UPDATE office.appointment
SET    startTime = '20110712 15:30',
       endTime = '20110712 15:59:59'
WHERE  appointmentId = 1;
```

which fails like it should.

www.it-ebooks.info

```
Msg 50000, Level 16, State 16, Procedure appointment$insertAndUpdateTrigger, Line 38
Appointment for doctor 00001 overlapped existing appointment
```

If this seems like a lot of work, it kind of is. And in reality, whether or not you actually implement this solution in a trigger is going to be determined by exactly what it is you are doing. However, the techniques of checking range uniqueness can clearly be useful if only to check existing data is correct, because in some cases, what you may want to do is to let data exist in intermediate states that aren't pristine and then write checks to "certify" that the data is correct before closing out a day. For a doctor's office, this might involve prioritizing certain conditions above other appointments, so a checkup gets bumped for a surgery. Daily, a query may be executed by the administrative assistant at the close of the day to clear up any scheduling issues.

Approximate Uniqueness

The most difficult case of uniqueness is actually quite common, and it is usually the most critical to get right. It is also a topic far too big to cover with a coded example, because in reality, it is more of a political question than a technical one. For example, if two people call in to your company from the same phone number and say their name is Louis Davidson, are they the same person? Whether you can call them the same person is a very important decision and one that is based largely on the industry you are in, made especially tricky due to privacy laws (if you give one person who claims to be Louis Davidson the data of the real Louis Davidson, well, that just isn't going to be good). I don't talk much about privacy laws in this book, mostly because that subject is very messy, but also because dealing with privacy concerns is:

- Largely just an extension of the principles I have covered so far, and will cover in the next chapter on security.
- Widely varied by industry and type of data you need to store

The principles of privacy are part of what makes the process of identification so difficult. At one time, companies would just ask for a customer's social security number and use that as identification in a very trusting manner. Of course, no sooner does some value become used widely by lots of organizations than it begins to be abused. So the goal of your design is to work at getting your customer to use a number to identify themselves to you. This customer number will be used as a login to the corporate web site, for the convenience card that is being used by so many businesses, and also likely on any correspondence. The problem is how to gather this information. When a person calls a bank or doctor, the staff member answering the call always asks some random questions to better identify the caller. For many companies, it is impossible to force the person to give information, so it is not always possible to force customers to uniquely identify themselves. You can entice them to identify themselves, such as by issuing a customer savings card, or you can just guess from bits of information that can gathered from a web browser, telephone number, and so on.

So the goal becomes to match people to the often-limited information they are willing to provide. Generally speaking, you can try to gather as much information as possible from people, such as

- Name
- Address, even partial
- Phone Number(s)
- Payment method
- E-mail address(es)

www.it-ebooks.info

And so on. Then, depending on the industry, you determine levels of matching that work for you. Lots of methods and tools are available to you, from standardization of data to make direct matching possible, fuzzy matching, and even third-party tools that will help you with the matches. The key, of course, is that if you are going to send a message alerting of a sale to repeat customers, only a slight bit of a match might be necessary, but if you are sending personal information, like how much money they have spent, a very deterministic match ought to be done. Identification of multiple customers in your database that are actually the same is the holy grail of marketing, but it is achievable given you respect your customer's privacy and use their data in a safe manner.

Data-Driven Design

One of the worst practices I see some programmers do is get in the habit of programming using specific keys to force a specific action. For example, they will get requirements that specify that for customer's 1 and 2, we need to do action A, and customer 3 do action B. So they go in and code:

```
IF @customerId in ('1', '2')
   Do ActionA(@customerId)
ELSE IF @customerId in ('3')
   Do ActionB(@customerId)
```

It works, so they breathe a sigh of relief and move on. But the next day, they get a request that customer 4 should be treated in the same manner as customer 3. They don't have time to do this request immediately because it requires a code change, which requires testing. So over the next month, they add `'4'` to the code, test it, deploy it, and claim it required 40 hours of programming time.

This is clearly not optimal, so the next best thing is to determine why we are doing `ActionA` or `ActionB`. We might determine that for `CustomerType: 'Great'`, we do `ActionA`, but for `'Good'`, we do `ActionB`. So you could code

```
IF @customerType = 'Great'
   Do ActionA(@customerId)
ELSE IF @customerType = 'Good'
   Do ActionB(@customerId)
```

Now adding another customer to these groups is a fairly simple case. You set the `customerType` value to `Great` or `Good`, and one of these actions occurs in you code automatically. But (as you might hear on any infomercial) you can do better! The shortcoming in this design is now how do you change the treatment of good customers if you want to have them do `ActionA` temporarily? In some cases, the answer is to add to the definition of the `CustomerType` table and add a column to indicate what action to take. So you might code:

```
CREATE TABLE CustomerType
(
        CustomerType varchar(20) NOT NULL CONSTRAINT PKCustomerType PRIMARY KEY,
        ActionType   char(1) NOT NULL CONSTRAINT CHKCustomerType_ActionType_Domain
                                    CHECK (CustomerType in ('A','B'))
);
```

Now, the treatment of this `CustomerType` can be set at any time to whatever the user decides. The only time you may need to change code (requiring testing, downtime, etc.) is if you need to change what an action means or add a new one. Adding different types of customers, or even changing existing ones would be a nonbreaking change, so no testing is required.

The basic goal should be that the structure of data should represent the requirements, so rules are enforced by varying data, not by having to hard-code special cases. In our previous example, you could create an override

www.it-ebooks.info

at the customer level by adding the `ActionType` to the customer. Flexibility at the code level is ultra important, particularly to your support staff. In the end, the goal of a design should be that changing configuration should not require code changes, so create attributes that will allow you to configure your data and usage.

■ **Note** In the code project part of the downloads for this chapter, you will find a coded example of data-driven design that demonstrates these principals in a complete, SQL coded solution.

Hierarchies

Hierarchies are a peculiar topic in relational databases. Hierarchies happen everywhere in the "real" world, starting with a family tree, corporation organizational charts, species charts, and parts breakdowns. Even the Lego example from earlier in this chapter, if modeled to completion, would include a hierarchy for sets as sometimes sets are parts of other sets to create a complete bill of materials for any set. Structure-wise, there are two sorts of hierarchies you will face in the real world, a tree structure, where every item can have only one parent, and graphs, where you can have more than one parent in the structure.

The challenge is to implement hierarchies in such a manner that they are optimal for your needs, particularly as they relate to the operations of your OLTP database. In this section, we will quickly go over the two major methods for implementing hierarchies that are the most common for use in SQL Server:

- Self referencing/recursive relationship/adjacency list
- Using the `HierarchyId` datatype to implement a tree structure

Finally, we'll take a brief architectural overview of a few other methods made popular by a couple of famous data architects; these methods can be a lot faster to use but require a lot more overhead to maintain, but sometimes, they're just better when your hierarchy is static and you need to do a lot of processing or querying.

Self Referencing/Recursive Relationship/Adjacency List

The self-referencing relationship is definitely the easiest method to implement a hierarchy for sure. We covered it a bit back in Chapter 3 when we discussed recursive relationships. They are recursive in nature because of the way they are worked with, particularly in procedural code. In relational code, you use a form of recursion where you fetch the top-level nodes, then all of their children, then the children of their children, and so on. In this section, I will cover trees (which are single parent hierarchies) and then graphs, which allow every node to have multiple parents.

Trees (Single-Parent Hierarchies)

To get started, I will create a table that implements a corporate structure with just a few basic attributes, including a self-referencing column. The goal will be to implement a corporate structure like the one shown in Figure 8-3.

The most important thing to understand when dealing with trees in SQL Server is that the most efficient way to work with trees in a procedural language is not the most efficient way to work with data in a set-based relational language. For example, if you were searching a tree in a functional language, you would likely use a recursive algorithm where you traverse the tree one node at a time, from the topmost item, down to the

www.it-ebooks.info

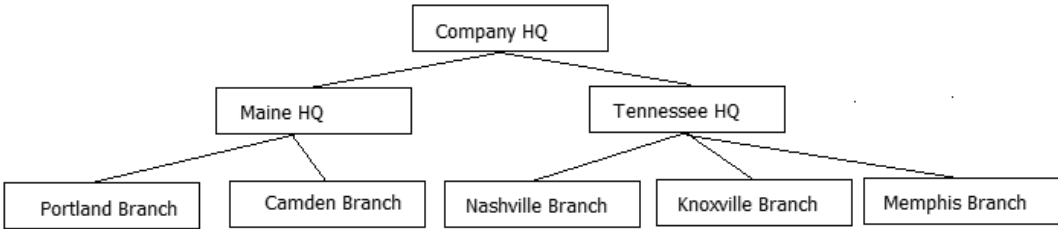


Figure 8-3. *Demonstration company hierarchy*

lowest in the tree, and then work your way around to all of the nodes. In Figure 8-4, I show this for the left side of the tree.

This is referred to as a depth-first search and is really fast when the language is optimized for single-instance-at-a-time access, particularly when you can load the entire tree into RAM. If you attempted to implement this using T-SQL, you would find that it is very slow, as most any iterative processing can be. In SQL, we use what is called a breadth-first search that can be scaled to many more nodes, because the number of queries is limited to the number of levels in the hierarchy. The limitations here pertain to the size of the temporary storage needed and how many rows you end up with on each level. Joining to an unindexed temporary set is bad in your code, and it is not good in SQL Server's algorithms either.

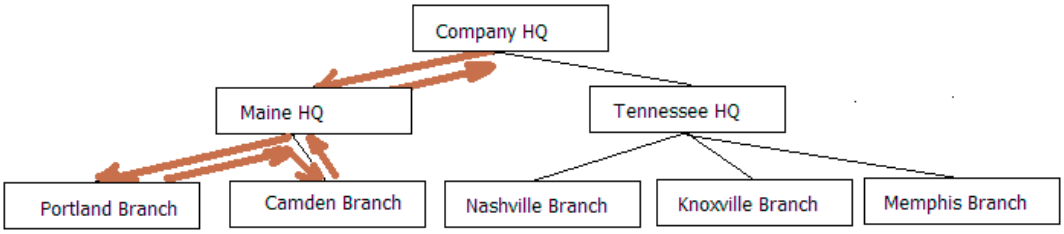


Figure 8-4. *Sample tree structure searched depth first*

A tree can be broken down into levels, from the parent row that you are interested in. From there, the levels increase as you are one level away from the parent, as shown in Figure 8-5.

Now, working with this structure will deal with each level as a separate set, joined to the matching results from the previous level. You iterate one level at a time, matching rows from one level to the next. This reduces the number of queries to use the data down to three, rather than a minimum of eight, plus the overhead of going back and forth from parent to child. To demonstrate working with adjacency list tables, let's create a table to

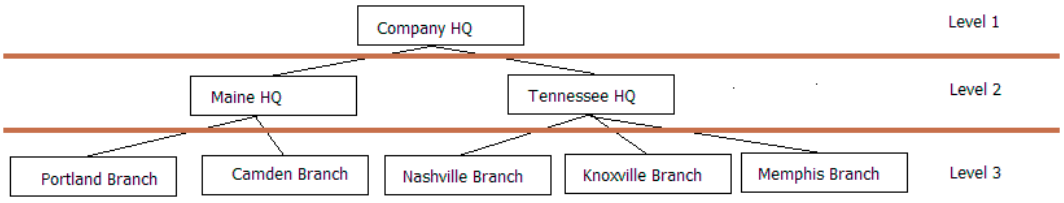


Figure 8-5. *Sample tree structure with levels*

www.it-ebooks.info

represent a hierarchy of companies that are parent to one another. The goal of our table will be to implement the structure, as shown in Figure 8-6.

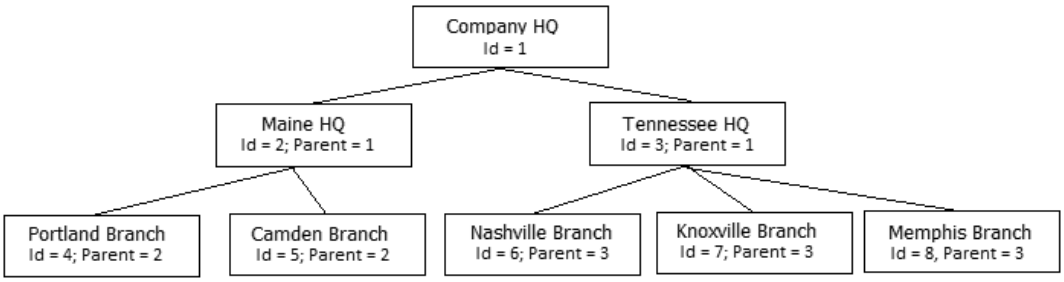


Figure 8-6. *Diagram of basic adjacency list*

So we will create the following table:

```
CREATE SCHEMA corporate;
GO
CREATE TABLE corporate.company
(
        companyId int NOT NULL CONSTRAINT PKcompany primary key,
        name varchar(20) NOT NULL CONSTRAINT AKcompany_name UNIQUE,
        parentCompanyId int NULL
          CONSTRAINT company$isParentOf$company REFERENCES corporate.company(companyId)
);
```

Then, I will load data to set up a table like the graphic in Figure 8-3:

```
INSERT INTO corporate.company (companyId, name, parentCompanyId)
VALUES  (1, 'Company HQ', NULL),
        (2, 'Maine HQ',1),          (3, 'Tennessee HQ',1),
        (4, 'Nashville Branch',3), (5, 'Knoxville Branch',3),
        (6, 'Memphis Branch',3),   (7, 'Portland Branch',2),
        (8, 'Camden Branch',2);
```

Now, taking a look at the data

```
SELECT *
FROM   corporate.company;
```

returns:

```
companyId       name                parentCompanyId
---------       ----------------    ---------------
1               Company HQ          NULL
2               Maine HQ            1
3               Tennessee HQ        1
4               Nashville Branch    3
5               Knoxville Branch    3
6               Memphis Branch      3
```


www.it-ebooks.info

```
7               Portland Branch       2
8               Camden Branch         2
```

Now, dealing with this data in a hierarchical manner is pretty simple. In the next code, we will write a query to get the children of a given node and add a column to the output that shows the hierarchy. I have commented the code to show what I was doing, but it is fairly straightforward how this code works:

```
--getting the children of a row (or ancestors with slight mod to query)
DECLARE @companyId int = <set me>;

;WITH companyHierarchy(companyId, parentCompanyId, treelevel, hierarchy)
AS
(
        --gets the top level in hierarchy we want. The hierarchy column
        --will show the row's place in the hierarchy from this query only
        --not in the overall reality of the row's place in the table
        SELECT  companyID, parentCompanyId,
                1 as treelevel, CAST(companyId AS varchar(max)) AS hierarchy
        FROM    corporate.company
        WHERE   companyId=@companyId

        UNION ALL

        --joins back to the CTE to recursively retrieve the rows
        --note that treelevel is incremented on each iteration
        SELECT  company.companyID, company.parentCompanyId,
                treelevel + 1 AS treelevel,
                hierarchy + '\' +cast(company.companyId AS varchar(20)) AS hierarchy
        FROM corporate.company
                INNER JOIN companyHierarchy
                --use to get children
                ON company.parentCompanyId= companyHierarchy.companyID
                --use to get parents
                --ON company.CompanyId= companyHierarchy.parentcompanyID
)
--return results from the CTE, joining to the company data to get the
--company name
SELECT  company.companyID,company.name,
        companyHierarchy.treelevel, companyHierarchy.hierarchy
FROM    corporate.company
            INNER JOIN companyHierarchy
                 ON company.companyID = companyHierarchy.companyID
ORDER BY hierarchy;
```

Running this code with `@companyId = 1`, you will get the following:

```
companyID   name              treelevel   hierarchy
---------   ---------------   ---------   ----------
1           Company HQ        1           1
2           Maine HQ          2           1\2
```


www.it-ebooks.info

```
7           Portland Branch    3           1\2\7
8           Camden Branch      3           1\2\8
3           Tennessee HQ       2           1\3
4           Nashville Branch   3           1\3\4
5           Knoxville Branch   3           1\3\5
6           Memphis Branch     3           1\3\6
```

■ **Tip** Make a note of the hierarchy output here. This is very similar the data used by the path method and will show up in the `hierarchyId` examples as well.

The hierarchy column shows you the position of each of the children of the `'Company HQ'` row, and since this is the only row with a null `parentCompanyId`, you don't have to start at the top; you can start in the middle. For example, the `'Tennessee HQ'(@companyId = 3)` row would return

```
companyID   name               treelevel   hierarchy
---------   ----------------   ---------   -----------
3           Tennessee HQ       1           3
4           Nashville Branch   2           3\4
5           Knoxville Branch   2           3\5
6           Memphis Branch     2           3\6
```

If you want to get the parents of a row, you need to make just a small change to the code. Instead of looking for rows in the CTE that match the `companyId` of the `parentCompanyId`, you look for rows where the `parentCompanyId` in the CTE matches the `companyId`. I left in some code with comments:

```
--use to get children
ON company.parentCompanyId= companyHierarchy.companyID
--use to get parents
--ON company.CompanyId= companyHierarchy.parentcompanyID
```

Comment out the first `ON`, and uncomment the second one:

```
--use to get children
--ON company.parentCompanyId= companyHierarchy.companyID
--use to get parents
ON company.CompanyId= companyHierarchy.parentcompanyID
```

And set `@companyId` to a row with parents, such as 4. Running this you will get

```
companyID   name               treelevel   hierarchy
---------   ----------------   ---------   ---------
4           Nashville Branch   1           4
3           Tennessee HQ       2           4\3
1           Company HQ         3           4\3\1
```


www.it-ebooks.info

The hierarchy column now shows the relationship of the row to the starting point in the query, not it's place in the tree. Hence, it seems backward, but thinking back to the breadth first searching approach, you can see that on each level, the hierarchy columns in all examples have added data for each iteration.

I should also make note of one issue with hierarchies, and that is circular references. We could easily have the following situation occur:

```
ObjectId      ParentId
--------      --------
1             3
2             1
3             2
```

In this case, anyone writing a recursive type query would get into an infinite loop because every row has a parent, and the cycle never ends. This is particularly dangerous if you limit recursion on a CTE (via the MAXRECURSION hint) and you stop after *N* iterations rather than failing, and hence never noticing.

Graphs (Multiparent Hierarchies)

Querying graphs (and in fact, hierarchies as well) are a very complex topic that is well beyond the scope of this book and chapter. It is my goal at this point to demonstrate how to model and implement graphs and leave the job of querying them to an advanced query book.

The most common example of a graph is a product breakdown. Say you have part A, and you have two assemblies that use this part. So the two assemblies are parents of part A. Using an adjacency list embedded in the table with the data you cannot represent anything other than a tree. We split the data from the implementation of the hierarchy. As an example, consider the following schema with parts and assemblies.

First, we create a table for the parts:

```
CREATE SCHEMA Parts;
GO
CREATE TABLE Parts.Part
(
        PartId int NOT NULL CONSTRAINT PKPartsPart PRIMARY KEY,
        PartNumber char(5) NOT NULL CONSTRAINT AKPartsPart UNIQUE,
        Name varchar(20) NULL
);
```

Then, we load in some simple data:

```
INSERT  INTO Parts.Part (PartId, PartNumber,Name)
VALUES  (1,'00001','Screw'),(2,'00002','Piece of Wood'),
        (3,'00003','Tape'),(4,'00004','Screw and Tape'),
        (5,'00005','Wood with Tape');
```

Next, a table to hold the part containership setup:

```
CREATE TABLE Parts.Assembly
(
        PartId int NOT NULL
          CONSTRAINT FKPartsAssembly$contains$PartsPart
                REFERENCES Parts.Part(PartId),
        ContainsPartId int NOT NULL
          CONSTRAINT FKPartsAssembly$isContainedBy$PartsPart
```


www.it-ebooks.info

```
            REFERENCES Parts.Part(PartId),
        CONSTRAINT PKPartsAssembly PRIMARY KEY (PartId, ContainsPartId),
);
```

Now, you can load in the data for the Screw and Tape part, by making the part with partId 4 a parent to 1 and 3:

```
INSERT INTO PARTS.Assembly(PartId,ContainsPartId)
VALUES (4,1),(4,3);
```

Next, you can do the same thing for the Wood with Tape part:

```
INSERT INTO Parts.Assembly(PartId,ContainsPartId)
VALUES (5,1),(4,2);
```

Using a graph can be simplified by dealing with each individual tree independently of one another by simply picking a parent and delving down. Cycles should be avoided, but it should be noted that the same part could end up being used at different levels in the hierarchy. The biggest issue is making sure that you don't double count data because of the parent to child cardinality that is greater than 1. Graph coding is a very complex topic that I won't go into here in any depth, while modeling them is relatively straightforward.

Implementing the Hierarchy Using the hierarchyTypeId Type

In SQL Server 2008, Microsoft added a new datatype called hierarchyTypeId. It is used to do some of the heavy lifting of dealing with hierarchies. It has some definite benefits in that it makes queries on hierarchies fairly easier, but it has some difficulties as well.

The primary downside to the hierarchyId datatype is that it is not as simple to work with for some of the basic tasks as is the self-referencing column. Putting data in this table will not be as easy as it was for that method (recall all of the data was inserted in a single statement, this will not be possible for the hierarchyId solution). However, on the bright side, the types of things that are harder with using a self-referencing column will be notably easier, but some of the hierarchyId operations are not what you would consider natural at all.

As an example, I will set up an alternate company table named corporate2 where I will implement the same table as in the previous example using hierarchyId instead of the adjacency list.a hierarchy of companies:

```
CREATE TABLE corporate.company2
(
  companyOrgNode hierarchyId NOT NULL
        CONSTRAINT AKcompany UNIQUE,
  companyId int NOT NULL CONSTRAINT PKcompany2 primary key,
  name      varchar(20) NOT NULL CONSTRAINT AKcompany2_name UNIQUE,
);
```

To insert a root node (with no parent), you use the GetRoot() method of the hierarchyId type without assigning it to a variable:

```
INSERT corporate.company2 (companyOrgNode, CompanyId, Name)
VALUES (hierarchyid::GetRoot(), 1, 'Company HQ');
```

To insert child nodes, you need to get a reference to the parentCompanyOrgNode that you want to add, then find its child with the largest companyOrgNode value, and finally, use the getDecendant() method of the companyOrgNode to have it generate the new value. I have encapsulated it into the following procedure (based on

www.it-ebooks.info

the procedure in the tutorials from books online, with some additions to support root nodes and single threaded inserts, to avoid deadlocks and/or unique key violations), and comments to explain how the code works:

```
CREATE PROCEDURE corporate.company2$insert(@companyId int, @parentCompanyId int,
                                           @name varchar(20))
AS
BEGIN
    SET NOCOUNT ON
    --the last child will be used when generating the next node,
    --and the parent is used to set the parent in the insert
    DECLARE @lastChildofParentOrgNode hierarchyid,
            @parentCompanyOrgNode hierarchyid;
    IF @parentCompanyId IS NOT NULL
      BEGIN
            SET @parentCompanyOrgNode =
                    ( SELECT companyOrgNode
                      FROM corporate.company2
                      WHERE companyID = @parentCompanyId)
            IF @parentCompanyOrgNode IS NULL
                BEGIN
                      THROW 50000, 'Invalid parentCompanyId passed in',16;
                      RETURN -100;
            END
    END

    BEGIN TRANSACTION;

            --get the last child of the parent you passed in if one exists
            SELECT @lastChildofParentOrgNode = max(companyOrgNode)
            FROM corporate.company2 (UPDLOCK) --compatibile with shared, but blocks
                                      --other connections trying to get an UPDLOCK
            WHERE companyOrgNode.GetAncestor(1) =@parentCompanyOrgNode ;

            --getDecendant will give you the next node that is greater than
            --the one passed in. Since the value was the max in the table, the
            --getDescendant Method returns the next one
            INSERT corporate.company2 (companyOrgNode, companyId, name)
              --the coalesce puts the row as a NULL this will be a root node
              --invalid parentCompanyId values were tossed out earlier
            SELECT COALESCE(@parentCompanyOrgNode.GetDescendant(
                    @lastChildofParentOrgNode, NULL),hierarchyid::GetRoot())
                    ,@companyId, @name;
    COMMIT;
END
```

Now, create the rest of the rows:

```
--exec corporate.company2$insert @companyId = 1, @parentCompanyId = NULL,
--                                @name = 'Company HQ'; --already created
exec corporate.company2$insert @companyId = 2, @parentCompanyId = 1,
                               @name = 'Maine HQ';
exec corporate.company2$insert @companyId = 3, @parentCompanyId = 1,
```


www.it-ebooks.info

```
                                  @name = 'Tennessee HQ';
exec corporate.company2$insert @companyId = 4, @parentCompanyId = 3,
                                  @name = 'Knoxville Branch';
exec corporate.company2$insert @companyId = 5, @parentCompanyId = 3,
                                  @name = 'Memphis Branch';
exec corporate.company2$insert @companyId = 6, @parentCompanyId = 2,
                                  @name = 'Portland Branch';
exec corporate.company2$insert @companyId = 7, @parentCompanyId = 2,
                                  @name = 'Camden Branch';
```

You can see the data in its raw format here:

```
SELECT  companyOrgNode, companyId, name
FROM    corporate.company2;
```

This returns a fairly uninteresting result set, particularly since the companyOrgNode value is useless in this untranslated format:

```
companyOrgNode    companyId    name
--------------    ---------    ----------------
0x                1            Company HQ
0x58              2            Maine HQ
0x68              3            Tennessee HQ
0x6AC0            4            Knoxville Branch
0x6B40            5            Nashville Branch
0x6BC0            6            Memphis Branch
0x5AC0            7            Portland Branch
0x5B40            8            Camden Branch
```

But this is not the most interesting way to view the data. The type includes methods to get the level, the hierarchy, and more:

```
SELECT  companyId, companyOrgNode.GetLevel() AS level,
        name, companyOrgNode.ToString() AS hierarchy
FROM    corporate.company2;
```

which can be really useful in queries:

```
companyId    level     name               hierarchy
---------    -----     ----------------   ---------
1            0         Company HQ         /
2            1         Maine HQ           /1/
3            1         Tennessee HQ       /2/
4            2         Knoxville Branch   /2/1/
5            2         Memphis Branch     /2/2/
6            2         Portland Branch    /1/1/
7            2         Camden Branch      /1/2/
```

Getting all of the children of a node is far easier than it was with the previous method. The hierarchyId type has an IsDecendantOf method you can use. For example, to get the children of companyId = 3, use the following:

www.it-ebooks.info

```
DECLARE @companyId int = 3;
SELECT Target.companyId, Target.name, Target.companyOrgNode.ToString() AS hierarchy
FROM   corporate.company2 AS Target
        JOIN corporate.company2 AS SearchFor
                ON SearchFor.companyId = @companyId
                        AND Target.companyOrgNode.IsDescendantOf
                                (SearchFor.companyOrgNode) = 1;
```

This returns

```
companyId     name                hierarchy
---------     ----------------    ---------
3             Tennessee HQ        /2/
4             Knoxville Branch    /2/1/
5             Memphis Branch      /2/2/
```

What is nice is that you can see in the hierarchy the row's position in the overall hierarchy without losing how it fits into the current results. In the opposite direction, getting the parents of a row isn't much more difficult. You basically just switch the position of the `SearchFor` and the `Target` in the `ON` clause:

```
DECLARE @companyId int = 3;
SELECT Target.companyId, Target.name, Target.companyOrgNode.ToString() AS hierarchy
FROM   corporate.company2 AS Target
         JOIN corporate.company2 AS SearchFor
                ON SearchFor.companyId = @companyId
                        AND SearchFor.companyOrgNode.IsDescendantOf
                                (Target.companyOrgNode) = 1;
```

This returns

```
companyId     name            hierarchy
---------     ------------    ---------
1             Company HQ      /
3             Tennessee HQ    /2/
```

This query is a bit easier to understand than the recursive CTEs we previously needed to work with. And this is not all that the datatype gives you. This chapter and section are meant to introduce topics, not be a complete reference. Check out Books Online for a full reference to `hierarchyId`.

However, while some of the usage is easier, using `hierarchyId` some negatives, most particularly when moving a node from one parent to another. There is a reparent method for `hierarchyId`, but it only works on one node at a time. To reparent a row (if, say, Oliver is now reporting to Cindy rather than Bobby), you will have to reparent all of the people that work for Oliver as well. In the adjacency model, simply moving modifying one row can move all rows at once.

Alternative Methods/Query Optimizations

Dealing with hierarchies in relational data has long been a well trod topic. As such, a lot has been written on the subject of hierarchies and quite a few other techniques that have been implemented. In this section, I will give an overview of three other ways of dealing with hierarchies that have been and will continue to be used in designs:

www.it-ebooks.info

- *Path technique*: In this method, which is similar to using `hierarchyId`, you store the path from the child to the parent in a formatted text string.
- *Nested sets*: Use the position in the tree to allow you to get children or parents of a row very quickly.
- *Kimball helper table*: Basically, this stores a row for every single path from parent to child. It's great for reads but tough to maintain and was developed for read-only situations, like read-only databases.

Each of these methods has benefits. Each is more difficult to maintain than a simple adjacency model or even the `hierarchyId` solution but can offer benefits in different situations. In the following sections, I am going to give a brief illustrative overview of each. In the downloads for the book, each of these will have example code that is not presented in the book in a separate file from the primary chapter example file.

Path Technique

The path technique is pretty much the manual version of the hierarchy method. In it, you store the path from the child to the parent. Using our hierarchy that we have used so, to implement the path method, we could use the set of data in Figure 8-7. Note that each of the tags in the hierarchy will use the surrogate key for the key values in the path. In Figure 8-7, I have included a diagram of the hierarchy implemented with the path value set for our design.

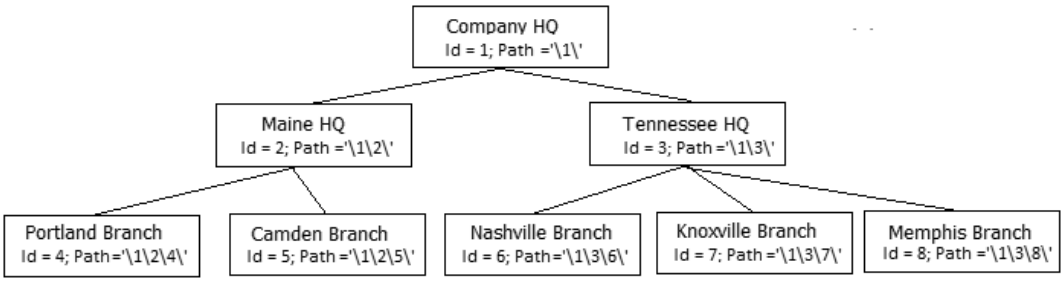


Figure 8-7. *Sample hierarchy diagram with values for the path technique*

With the path in this manner, you can find all of the children of a row using the path in a like expression. For example, to get the children of the `Main HQ` node, you can use a `WHERE` clause such as `WHERE Path LIKE '\1\2\%'` to get the children, and the path to the parents is directly in the path too. So the parents of the Portland Branch, whose path is `'\1\2\4\'` are `'\1\2\'` and `'\1\'`.

The path method has a bit of an issue with indexing, since you are constantly doing substrings. But they are usually substrings starting with the beginning of the string, so it can be fairly performant. Of course, you have to maintain the hierarchy manually, so it can be fairly annoying to use and maintain this method like this. Generally, `hierarchyId` seems to be a better fit since it does a good bit of the work for you rather than managing it yourself manually.

Nested Sets

One of the more clever methods was created in 1992 by Michael J. Kamfonas. It was introduced in an article named "Recursive Hierarchies: The Relational Taboo!" in The Relational Journal, October/November 1992. You can still find it on his web site, `www.kamfonas.com`. It is also a favorite of Joe Celko who has written a book about

www.it-ebooks.info

hierarchies named *Joe Celko's Trees and Hierarchies in SQL for Smarties* (Morgan Kaufmann, 2004); check it out for further reading about this and other types of hierarchies.

The basics of the method is that you organize the tree by including pointers to the left and right of the current node, enabling you to do math to determine the position of an item in the tree. Again, going back to our company hierarchy, the structure would be as shown in Figure 8-8:

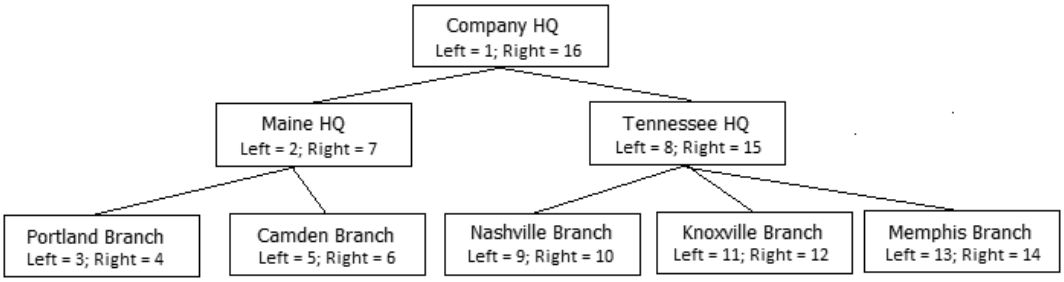


Figure 8-8. *Sample hierarchy diagram with values for the nested sests technique*

This has the value of now being able to determine children and parents of a node very quickly. To find the children of `Maine HQ`, you would say `WHERE Left > 2 and Right < 7`. No matter how deep the hierarchy, there is no traversing the hierarchy at all, just simple math. To find the parents of `Maine HQ`, you simple need to look for the case `WHERE Left < 2 and Right > 7`.

Adding a node has a slight negative effect of needing to update all rows to the right of the node, increasing their `Right` value, since every single row is a part of the structure. Deleting a node will require decrementing the `Right` value. Even reparenting becomes a math problem, just requiring you to update the linking pointers. Probably the biggest downside is that it is not a very natural way to work with the data, since you don't have a link directly from parent to child to navigate.

Kimball Helper Table

Finally, in a method that is going to be the most complex to manage (but in most cases, the fastest to query), you can use a method that Ralph Kimball created for dealing with hierarchies, particularly in a data warehousing/read-intensive setting, but it could be useful in an OLTP setting if the hierarchy is stable. Going back to our adjacency list implementation, shown in Figure 8-9, assume we already have this implemented in SQL.

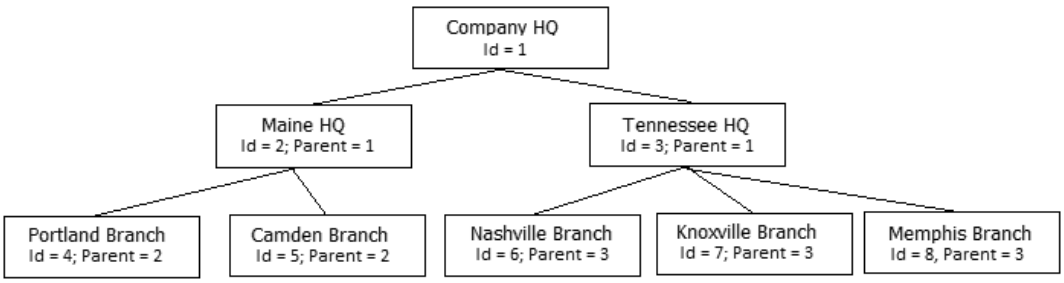


Figure 8-9. *Sample hierarchy diagram with values for the adjacency list technique repeated for the Kimball helper table method*

www.it-ebooks.info

To implement this method, you will use a table of data that describes the hierarchy with one row per parent to child relationship, for every level of the hierarchy. So there would be a row for `Company HQ` to `Maine HQ`, `Company HQ` to `Portland Branch`, etc. The helper table provides the details about distance from parent, if it is a root node or a leaf node. So, for the leftmost four items (`1`, `2`, `4`, `5`) in the tree, we would get the following table.

```
ParentId        ChildId         Distance        ParentRootNodeFlag      ChildLeafNodeFlag
--------        --------        --------        ------------------      -----------------
1               2               1               1                       0
1               4               2               1                       1
1               5               2               1                       1
2               4               1               0                       1
2               5               1               0                       1
```

The power of this technique is that now you can simply ask for all children of `1` by looking for `WHERE ParentId = 1`, or you can look for direct descendents of `2` by saying `WHERE ParentId = 2 AND Distance = 1`. And you can look for all leaf notes of the parent by querying `WHERE ParentId = 1 AND ChildLeafNode = 1`.

The obvious downfall of this method is simple. It is hideous to maintain if the structure is frequently modified. To be honest, Kimball's purpose for the method was to optimize relational usage of hierarchies in the data warehouse, which is maintained by ETL. For this sort of purpose, this method should be the quickest, because all queries will be almost completely based on simple relational queries. Of all of the methods, this one will be the most natural for users, while being the less desirable to the team that has to maintain the data.

Images, Documents, and Other Files, Oh My

Storing large binary objects, such PDFs, images, and really any kind of object you might find in your Windows file system is generally not the historic domain of the relational database. As time has passed, however, it is becoming more and more commonplace.

When discussing how to store large objects in SQL Server, generally speaking this would be in reference to data that is (obviously) large but usually in some form of binary format that is not naturally modified using common T-SQL statements, for example, a picture or a formatted document. Most of the time, this is not considering simple text data or even formatted, semistructured text, or even highly structured text such as XML. SQL Server has an `XML` type for storing XML data (including the ability to index fields in the XML document), and it also has `varchar(max)`/`nvarchar(max)` types for storing very large "plain" text data. Of course, sometimes, you will want to store text data in the form of a Windows text file to allow users to manage the data naturally. When deciding a way to store binary data in SQL Server, there are typically two ways that are available:

- Storing a path reference to the file data
- Storing the binaries using SQL Server's storage engine

Prior to 2008, the question was pretty easy to answer indeed. Almost always, the most reasonable solution was to store files in the file system and just store a reference to the data in a `varchar` column. In SQL Server 2008, Microsoft implemented a new type of binary storage called a *filestream*, which allows binary data to be stored in the file system as actual files, which makes accessing this data from a client much faster than if it were stored in a binary column in SQL Server. In SQL Server 2011, the picture improves even more to give you a method to store any file data in the server that gives you access to the data using what looks like a typical network share. In all cases, you can deal with the data in T-SQL as before, and even that may be improved, though you cannot do partial writes to the values like you can in a basic `varbinary(max)` column.

www.it-ebooks.info

In the 2005 edition of this book, I separated the choice between the two possible ways to store binaries into one main simple reason to choose one or the other: transactional integrity. If you required transaction integrity, you use SQL Server's storage engine, regardless of the cost you would incur. If transaction integrity isn't tremendously important, use the file system. For example, if you were just storing an image that a user could go out and edit, leaving it with the same name, the file system is perfectly natural. Performance was a consideration, but if you needed performance, you could write the data to the storage engine first and then regularly refresh the image to the file system and use it from a cache.

In SQL Server 2008, the choice was expanded to include not only basic `varbinary(max)` columns but now included what is known as a filestream columns. Filestream column data is saved in the filesystem as files, which can be very fast because you can access the data using a special windows share, but the accessor must be within a transaction at the time. It can be useful in some situations, but it requires external APIs to make it work in a natural manner. The setup is pretty easy; first, you will enable filestream access for the server. For details on this process, check the Books Online topic "Enable and Configure FILESTREAM." The basics are to enable go to SQL Server Configuration Manager and choose the SQL Server Instance in SQL Server Services. Open the properties, and choose the FILESTREAM tab, as shown in Figure 8-10.

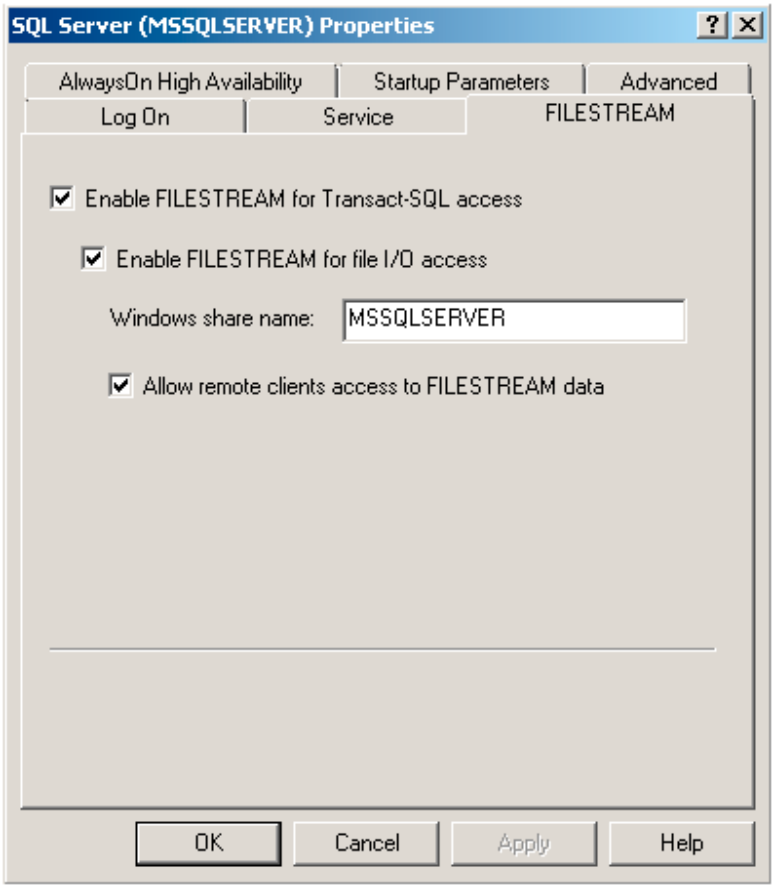


Figure 8-10. *Configuring the server for filestream access*

www.it-ebooks.info

The Windows share name will be used to access filetable data later in this chapter. Later in this section, there will be additional configurations based on how the filestream data will be accessed. Next, we create a sample database (instead of using pretty much any database as we have for the rest of this chapter).

```
CREATE DATABASE FileStorageDemo; --uses basic defaults from model databases
GO
USE FileStorageDemo;
GO
--will cover filegroups more in the chapter 10 on structures
ALTER DATABASE FileStorageDemo ADD
        FILEGROUP FilestreamData CONTAINS FILESTREAM;
```

■ **Tip** You cannot use filestream data in a database that also needs to use snapshot isolation level or that implements the `READ_COMMITTED_SNAPSHOT` database option.

Next, add a "file" to the database that is actually a directory for the filestream files:

```
ALTER DATABASE FileStorageDemo ADD FILE (
        NAME = FilestreamDataFile1,
        FILENAME = 'c:\sql\filestream') --directory cannot yet exist
TO FILEGROUP FilestreamData;
```

Now, you can create a table and include a `varbinary(max)` column with the keyword `FILESTREAM` after the datatype declaration. Note, too, that we need a unique identifier column with the `ROWGUIDCOL` property that is used by some of the system processes as a kind of special surrogate key.

```
CREATE TABLE TestSimpleFileStream
(
        TestSimpleFilestreamId INT NOT NULL
                CONSTRAINT PKTestSimpleFileStream PRIMARY KEY,
        FileStreamColumn VARBINARY(MAX) FILESTREAM NULL,
        RowGuid uniqueidentifier NOT NULL ROWGUIDCOL DEFAULT (NewId()) UNIQUE
        FILESTREAM_ON FilestreamData; --optional, goes to default otherwise
);
```

It is as simple as that. You can use the data exactly like it is in SQL Server, as you can create the data using as simple query:

```
INSERT INTO TestSimpleFileStream(TestSimpleFilestreamId,FileStreamColumn)
SELECT 1, CAST('This is an exciting example' AS varbinary(max));
```

and see it using a typical `SELECT`:

```
SELECT  TestSimpleFilestreamId,FileStreamColumn,
        CAST(FileStreamColumn AS varchar(40)) AS FileStreamText
FROM    TestSimpleFilestream;
```

I won't go any deeper into filestreams, because all of the more interesting bits of the technology from here are external to SQL Server in API code, which is well beyond the purpose of this section, which is to show you the basics of setting up the filestream column in your structures.

www.it-ebooks.info

In SQL Server 2012, we get a new feature for storing binary files called a filetable. A filetable is a special type of table that you can access using T-SQL or directly from the file system using the share we set up earlier in this section named MSSQLSERVER. One of the nice things for us is that we will actually be able to see the file that we create in a very natural manner.

The first thing you need to do is enable filetable access via Windows if it hasn't been done before (it won't error or anything if it has already been run):

```
EXEC sp_configure filestream_access_level, 2;
RECONFIGURE;
```

The domain for the parameter that is valued 2 (from Books Online) is 0: Disable filestream for the instance; 1: enable filestream for T-SQL; 2: Enable T-SQL and Win32 streaming access. Then, you enable and set up filetable style filestream in the database:

```
ALTER database FileStorageDemo
        SET FILESTREAM (NON_TRANSACTED_ACCESS = FULL,
                DIRECTORY_NAME = N'ProSQLServer2012DBDesign');
```

The setting NON_TRANSACTED_ACCESS lets you set if users can change data when accessing the data as a Windows share. The changes are not transactionally safe, so data stored in a filetable is not as safe as using a simple varbinary(max) or even one using the filestream attribute. It behaves pretty much like data on any file server, except that it will be backed up with the database, and you can easily associate a file with other data in the server using common relational constructs. The DIRECTORY_NAME parameter is there to add to the path you will access the data (this will be demonstrated later in this section).

The syntax for creating the filetable is pretty simple:

```
CREATE TABLE dbo.FileTableTest AS FILETABLE
  WITH (
        FILETABLE_DIRECTORY = 'FileTableTest',
        FILETABLE_COLLATE_FILENAME = database_default
        );
```

The FILETABLE_DIRECTORY is the final part of the path for access, and the FILETABLE_COLLATE_FILENAME determines the collation that the filenames will be treated as. It must be case insensitive, because Windows directories are case insensitive. I won't go in depth with all of the columns and settings, but suffice it to say that the filetable is based on a fixed table schema, and you can access it much like a common table. There are two types of rows, directories, and files. Creating a directory is easy. For example, if you wanted to create directory for Project 1:

```
INSERT INTO FiletableTest(name, is_directory)
VALUES ( 'Project 1', 1);
```

Then, you can view this data in the table:

```
SELECT  stream_id, file_stream, name
FROM    FileTableTest
WHERE   name = 'Project 1';
```

This will return (though with a different stream_id):

```
stream_id                            file_stream     name
-----------------------------------  ----------      ---------
9BCB8987-1DB4-E011-87C8-000C29992276 NULL            Project 1
```


www.it-ebooks.info

stream_id is automatically a unique key that you can relate to with your other tables, allowing you to simply present the user with a "bucket" for storing data. Note that the primary key of the table is the path_locator hierarchyId, but this is a changeable value. The stream_id value shouldn't ever change, though the file or directory could be moved. Before we go check it out in Windows, let's add a file to the directory. We will create a simple text file, with a small amount of text:

```
INSERT INTO FiletableTest(name, is_directory, file_stream)
VALUES ( 'Test.Txt', 0, CAST('This is some text' AS varbinary(max)));
```

Then, we can move the file to the directory we just created using the path_locator hierarchyId functionality:

```
UPDATE  FiletableTest
SET     path_locator = path_locator.GetReparentedValue( path_locator.GetAncestor(1),
          (SELECT path_locator FROM FiletableTest
            WHERE name = 'Project 1'
                AND parent_path_locator IS NULL
                AND is_directory = 1))
WHERE name = 'Test.Txt';
```

Now, go to the share that you have set up and view the directory in Windows. Using the function FileTableRootPath(), you can get the filetable path for the database, in our case, \\DENALI-PC\MSSQLSERVER\ProSQLServer2012DBDesign, which is my computer's name, the MSSQLSERVER we set up in Configuration Manager, and ProSQLServer2012DBDesign from the ALTER DATABASE statement turning on filestream.

Now, concatenating the root to the path for the directory, which can be retrieved from the file_stream column (yes, the value you see when querying it is NULL, which is a bit confusing). Now, execute this:

```
SELECT CONCAT(FileTableRootPath(),
         file_stream.GetFileNamespacePath()) AS FilePath;
FROM   dbo.FileTableTest
WHERE  name = 'Project 1'
         AND parent_path_locator IS NULL
         AND is_directory = 1;
```

This returns the following:

```
FilePath
------------------------------------------------------------------------
\\DENALI-PC\MSSQLSERVER\ProSQLServer2012DBDesign\FileTableTest\Project 1
```

You can then enter this into Explorer to see something like what's shown in Figure 8-11 (assuming you have everything configured correctly, of course). Note that security for the Windows share is the same as for the filetable through T-SQL, which you administer the same as with any regular table:

From here, I would suggest you drop a few files in the directory and check out the metadata for your files in your newly created filetable. It has a lot of possibilities. I will touch more on security in Chapter 9, but the basics are that security is based on Windows Authentication as you have it set up in SQL Server on the table, just like any other table.

www.it-ebooks.info

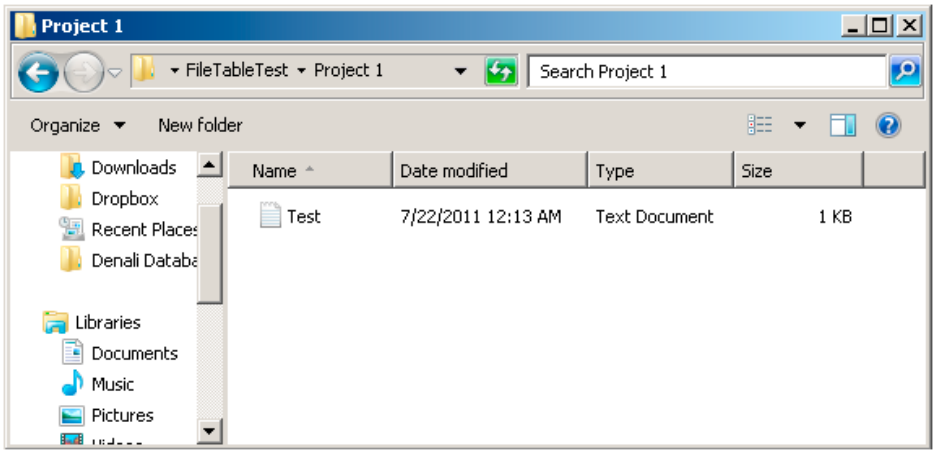


Figure 8-11. *Filetable directory opened in Windows Explorer*

■ **Note** If you try to use Notepad to access the text file on the same server as the share is located, you will receive an error due to the way notepad accesses files locally. Accessing the file from a remote location using Notepad will work fine.

I won't spend any more time covering the particulars of implementing with filetables. Essentially, with very little trouble, even a fairly basic programmer could provide a directory per client to allow the user to navigate to the directory and get the customer's associated files. And the files can be backed up with the normal backup operations.

So, mechanics aside, we have discussed essentially four different methods of storing binary data in SQL tables:

- Store a UNC path in a simple character column
- Store the binary data in a simple `varbinary(max)` column
- Store the binary data in a `varbinary(max)` using the filestream type
- Store the binary data using a filetable

■ **Tip** There is one other type of method of storing large binary values using what is called the Remote Blob Store API. It allows you to use an external storage device to store and manage the images. It is not a typical case, though it will definitely be of interest to people building high-end solutions needing to store blobs on an external device.

Each of these has some merit, and I will discuss the pros and cons in the following list. Like with any newer, seemingly easier technology, a filetable does feel like it might take the nod for a lot of upcoming uses, but definitely consider the other possibilities when evaluating your needs in the future.

www.it-ebooks.info

- *Transactional integrity*: It's far easier to guarantee that the image is stored and remains stored if it is managed by the storage engine, either as a filestream or as a binary value, than it is if you store the filename and path and have an external application manage the files. A filetable could be used to maintain transactional integrity, but to do so, you will need to disallow nontransaction modifications, which will then limit how much easier it is to work with.
- *Consistent backup of image and data*: Knowing that the files and the data are in sync is related to transactional integrity. Storing the data in the database, either as a binary columnar value or as a filestream/filetable, ensures that the binary values are backed up with the other database objects. Of course, this can cause your database size to grow very large, so it can be handy to use partial backups of just the data and not the images at times. Filegroups can be restored individually as well, but be careful not to give up integrity for faster backups if the business doesn't expect it.
- *Size*: For sheer speed of manipulation, for the typical object size less than 1MB, Books Online suggests using storage in a `varchar(max)`. If objects are going to be more than 2GB, you must use one of the filestream storage types.
- *API*: Which API is the client using? If the API does not support using the filestream type, you should definitely give it a pass. A filetable will let you treat the file pretty much like it was on any network share, but note that access/modification through the filesystem is not part of a transaction, so if you need the file modification to occur along with other changes as a transaction, it will not do.
- *Utilization*: How will the data be used? If it is used very frequently, then you would choose either filestream/filetable or file system storage. This could particularly be true for files that need to be read only. Filetable is a pretty nice way to allow the client to view a file in a very natural manner.
- *Location of files*: Filestream filegroups are located on the same server as the relational files. You cannot specify a UNC path to store the data. For filestream column use, the data, just like a normal filegroup, must be transactionally safe for utilization. (Again, as the preceding note states, there are devices that implement a remote blob store to get around this limitation.)
- *Encryption*: Encryption is not supported on the data store in filestream filegroups, even when transparent data encryption (TDE) is enabled.
- *Security*: If the image's integrity is important to the business process (such as the picture on a security badge that's displayed to a security guard when a badge is swiped), it's worth it to pay the extra price for storing the data in the database, where it's much harder to make a change. (Modifying an image manually in T-SQL is a tremendous chore indeed.) A filetable also has a disadvantage in that implementing row-level security (discussed in Chapter 9 in more detail) using views would not be possible, whereas when using a filestream-based column, you are basically using the data in a SQL-esque manner until you access the file, though still in the context of a transaction.

For three quick examples, consider a movie rental database. In one table, we have a `MovieRentalPackage` table that represents a particular packaging of a movie for rental. Because this is just the picture of a movie, it is a perfectly valid choice to store a path to the data. This data will simply be used to present electronic browsing of the store's stock, so if it turns out to not work one time, that is not a big issue. Have a column for the `PictureUrl` that is `varchar(200)`, as shown in Figure 8-12.

www.it-ebooks.info

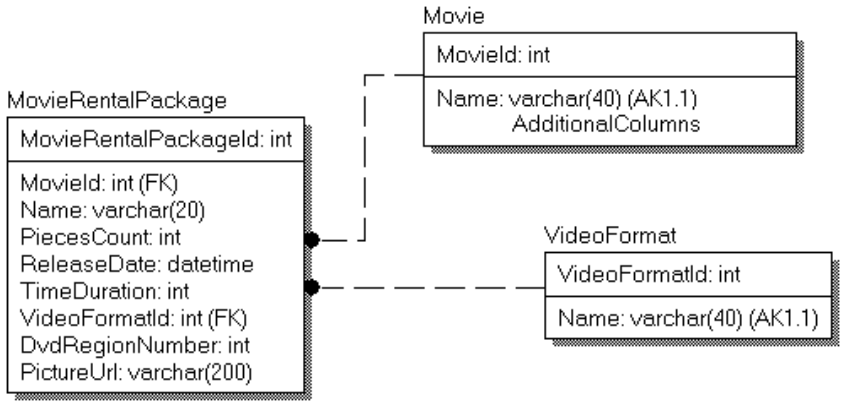


Figure 8-12. *MovieRentalPackage table with PictureUrl datatype set as a path to a file*

This path might even be on an Internet source where the filename is an HTTP:// address and be located in a web server's image cache and could be replicated to other web servers. The path may or may not be stored as a full UNC location; it really would depend on your infrastructure needs. The goal will be when the page is fetching data from the server to be able to build a bit of HTML such as this:

```
SELECT  '<img src = "' + MovieRentalPackage.PictureUrl + '">', …
FROM    MovieRentalPackage
WHERE   MovieId = @MovieId;
```

If this data were stored in the database as a binary format, it would need to be materialized onto disk as a file first and then used in the page, which is going to be far slower than doing it this way, no matter what your architecture. This is probably not a case where you would want to do this or go through the hoops necessary for filestream access, since transactionally speaking, if the picture link is broken, it would not invalidate the other data, and it is probably not very important. Plus, you will probably want to access this file directly, making the main web screens very fast and easy to code.

An alternative example might be accounts and associated users (see Figure 8-13). To fight fraud, a movie rental chain may decide to start taking customer pictures and comparing them whenever customers rent an item. This data is far more important from a security standpoint and has privacy implications. For this, I'll use a `varbinary(max)` for the person's picture in the database.

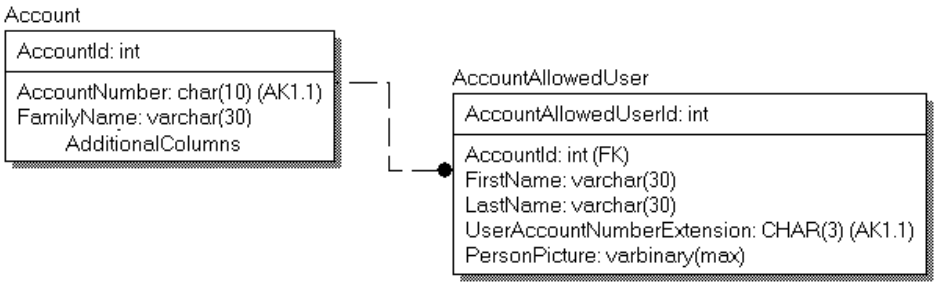


Figure 8-13. *Customer table with picture stored as data in the table*

www.it-ebooks.info

At this point, you have definitely decided that transactional integrity is necessary and that you want to retrieve the data directly from the server. The next thing to decide will be whether to employ filestreams. The big questions here would be if your API will support filestreams. If so, it would likely be a very good place to make use of them for sure. Size could play a part in the choice too, though security pictures would likely be less than 1MB anyhow.

Overall, speed probably isn't a big deal, and even if you needed to take the binary bits and stream them from SQL Server's normal storage into a file, it would probably still perform well enough since only one image needs to be fetched at a time, and performance will be adequate as long as the image displays before the rental transaction is completed. Don't get me wrong; the `varbinary(max)` types aren't that slow, but performance would be acceptable for these purposes even if they were.

Finally, consider if you wanted to implement a customer file system to store scanned images pertaining to the customer. Not enough significance is given to the data to require it to be managed in a structured manner, but they simply want to be able to create a directory to hold scanned data. The data does need to be kept in sync with the rest of the database. So you could extend your table to include a filetable (`AccountFileDirectory` in Figure 8-14, with `stream_id` modeled as primary, even though it is technically a unique constraint in implementation, you can reference an alternate key).

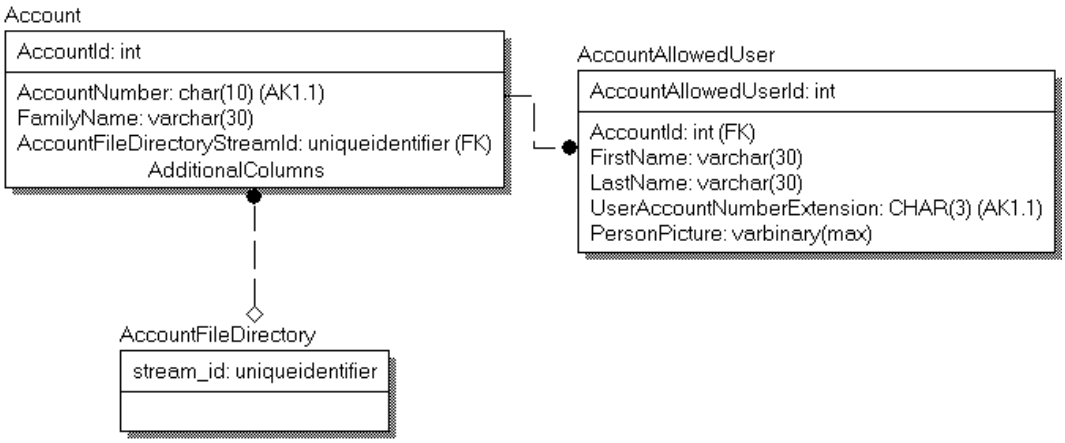


Figure 8-14. *Account model extended with an AccountFileDirectory*

In this manner, you have included a directory for the account's files that can be treated like a typical file structure but will be securely located with the account information. This will be not only very usable for the programmer and user alike but also give you the security of knowing the data is backed up with the account files and treated in the same manner as the account information.

One of the interesting changes in database design in SQL Server 2012 is dealing with unstructured file data, and it is a tremendously interesting change. Not everyone needs to store a lot of image data, but if you do, the new filetable feature definitely opens things up for much easier access.

Generalization

Very often, it is tempting to design more tables than is truly necessary for the system needs, because we get excited and start to design perfection. For example, if you were designing a database to store information about camp activities, it might be tempting to have an individual table for the archery class, the swimming class, and

www.it-ebooks.info

so on, modeling with great detail the activities of each camp activity. If there were 50 activities at the camp, you might have 50 tables, plus a bunch of other tables to tie these 50 together. In the end though, while these tables may not exactly look the same, you will start to notice that every table is used for basically the same thing. Assign an instructor, sign up kids to attend, and add a description. Rather than the system being about each activity, and so needing a model each of the different activities as being different from one another, what you truly needed was to model was the abstraction of a camp activity.

In the end, the goal is to consider where you can combine foundationally similar tables into a single table, particularly when multiple tables are constantly being treated as one, as you would have to do with the 50 camp activity tables, particularly to make sure kids weren't signing up their friends for every other session just to be funny.

During design, it is useful to look for similarities in utilization, columns, and so on, and consider collapsing multiple tables into one, ending up with a generalization/abstraction of what is truly needed to be modeled. Clearly, however, the biggest problem here is that sometimes you do need to store different information about some of the things your original tables were modeling. For example, if you needed special information about the snorkeling class, you might lose that if you just created one activity abstraction, and heaven knows the goal is *not* to end up with a table with 200 columns all prefixed with what ought to have been a table in the first place.

In those cases, you can consider using a subclassed entity for certain entities. Take the camp activity model. We include the generalized table for the generic `CampActivity`, which is where you would associate student and teachers who don't need special training, and in the subclassed tables, you would include specific information about the snorkeling and archery classes, likely along with the teachers who meet specific criteria, as shown in Figure 8-15.

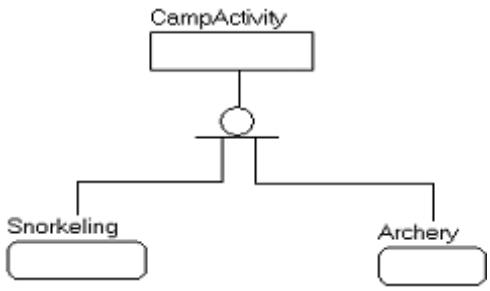


Figure 8-15. *Extending generalized entity with specific details as required*

As a coded example, we will look at a home inventory system. You have various types of stuff in your house, and you clearly want to inventory everything or at least everything valuable. So would you simply design a table for each type of item? That seems like too much trouble, because most everything you will simply have a description, picture, value, and a receipt. On the other hand, a single table, generalizing all of the items in your house down to a single list seems like it might not be enough for items that you need specific information about, like appraisals and serial numbers.

For example, some jewelry probably ought to be appraised and have the appraisal listed. Electronics and appliances ought to have brand, model, and alternatively, serial numbers captured. So the goal is to generalize a design to the level where you have a basic list of the home inventory that you can list, but you can also print a list of jewelry alone with extra detail or electronics with their information.

www.it-ebooks.info

So we implement the database as such. First, we create the generic table that holds generic item descriptions:

```
CREATE SCHEMA Inventory;
GO
CREATE TABLE Inventory.Item
(
        ItemId  int NOT NULL IDENTITY CONSTRAINT PKInventoryItem PRIMARY KEY,
        Name    varchar(30) NOT NULL CONSTRAINT AKInventoryItemName UNIQUE,
        Type    varchar(15) NOT NULL,
        Color   varchar(15) NOT NULL,
        Description varchar(100) NOT NULL,
        ApproximateValue numeric(12,2) NULL,
        ReceiptImage varbinary(max) NULL,
        PhotographicImage varbinary(max) NULL
);
```

I included two columns for holding an image of the receipt as well as a picture of the item. Like we discussed in the previous section, you might want to use a filetable construct to just allow various electronic items to be associated with this data, but it would probably be sufficient to simply have a picture of the receipt and the item minimally attached to the row for easy use. In the sample data, I always load the data with a simple hex value of 0x001 as a placeholder:

```
INSERT  INTO Inventory.Item
VALUES  ('Den Couch','Furniture','Blue','Blue plaid couch, seats 4',450.00,0x001,0x001),
        ('Den Ottoman','Furniture','Blue','Blue plaid ottoman that goes with couch',
         150.00,0x001,0x001),
        ('40 Inch Sorny TV','Electronics','Black',
         '40 Inch Sorny TV, Model R2D12, Serial Number XD49292',
         800,0x001,0x001),
        ('29 Inch JQC TV','Electronics','Black','29 Inch JQC CRTVX29 TV',800,0x001,0x001),
        ('Mom''s Pearl Necklace','Jewelery','White',
         'Appraised for $1300 in June of 2003. 30 inch necklace, was Mom''s',
         1300,0x001,0x001);
```

Checking out the data using the following query:

```
SELECT  Name, Type, Description
FROM    Inventory.Item;
```

We see that we have a good little system, though data isn't really organized how we need it, because in realistic usage, we will probably need some of the specific data from the descriptions:

```
Name                    Type            Description
--------------------    -----------     ----------------------------------
Den Couch               Furniture       Blue plaid couch, seats 4
Den Ottoman             Furniture       Blue plaid ottoman that goes with …
40 Inch Sorny TV        Electronics     40 Inch Sorny TV, Model R2D12, Ser…
29 Inch JQC TV          Electronics     29 Inch JQC CRTVX29 TV
Mom's Pearl Necklace    Jewelery        Appraised for $1300 in June of 200…
```


www.it-ebooks.info

At this point, we look at our data and reconsider the design. The two pieces of furniture are fine. We have a picture and a brief description. For the other two items, however, using the data becomes trickier. For electronics, the insurance company is going to want model and serial number for each, but the two TV entries use different formats, and one of them doesn't capture the serial number. Did they forget to capture it? Or does it not exist?

So, we add subclasses for cases where we need to have more information, to help guide the user as to how to enter data:

```
CREATE TABLE Inventory.JeweleryItem
(
        ItemId int CONSTRAINT PKInventoryJewleryItem PRIMARY KEY
          CONSTRAINT FKInventoryJewleryItem$Extends$InventoryItem
                REFERENCES Inventory.Item(ItemId),
        QualityLevel    varchar(10) NOT NULL,
        AppraiserName   varchar(100) NULL,
        AppraisalValue  numeric(12,2) NULL,
        AppraisalYear   char(4) NULL
);
GO
CREATE TABLE Inventory.ElectronicItem
(
        ItemId int CONSTRAINT PKInventoryElectronicItem PRIMARY KEY
                CONSTRAINT FKInventoryElectronicItem$Extends$InventoryItem
                        REFERENCES Inventory.Item(ItemId),
        BrandName varchar(20) NOT NULL,
        ModelNumber varchar(20) NOT NULL,
        SerialNumber varchar(20) NULL
);
```

Now, we adjust the data in the tables to have names that are meaningful to the family, but we can create views of the data that have more or less technical information to provide to other people—first, the two TVs. Note that we still don't have a serial number, but now, it would be simple to find electronics that have no serial number to tell us that we need to get one:

```
UPDATE  Inventory.Item
SET     Description = '40 Inch TV'
WHERE   Name = '40 Inch Sorny TV';
GO
INSERT  INTO Inventory.ElectronicItem (ItemId, BrandName, ModelNumber, SerialNumber)
SELECT  ItemId, 'Sorny','R2D12','XD49393'
FROM    Inventory.Item
WHERE   Name = '40 Inch Sorny TV';
GO
UPDATE  Inventory.Item
SET     Description = '29 Inch TV'
WHERE   Name = '29 Inch JQC TV';
GO
INSERT  INTO Inventory.ElectronicItem(ItemId, BrandName, ModelNumber, SerialNumber)
SELECT  ItemId, 'JVC','CRTVX29',NULL
FROM    Inventory.Item
WHERE   Name = '29 Inch JQC TV';
```

Finally, we do the same for the jewelry items, adding the appraisal value from the text.

www.it-ebooks.info

```
UPDATE  Inventory.Item
SET     Description = '30 Inch Pearl Neclace'
WHERE   Name = 'Mom''s Pearl Necklace';
GO
INSERT  INTO Inventory.JeweleryItem (ItemId, QualityLevel, AppraiserName,
        AppraisalValue,AppraisalYear)
SELECT  ItemId, 'Fine','Joey Appraiser',1300,'2003'
FROM    Inventory.Item
WHERE   Name = 'Mom''s Pearl Necklace';
```

Looking at the data now, you see the more generic list with names that are more specifically for the person maintaining the list:

```
SELECT  Name, Type, Description
FROM    Inventory.Item;
```

This returns

```
Name                 Type           Description
-------------------  -----------    ------------------------------
Den Couch            Furniture      Blue plaid couch, seats 4
Den Ottoman          Furniture      Blue plaid ottoman that goes w…
40 Inch Sorny TV     Electronics    40 Inch TV
29 Inch JQC TV       Electronics    29 Inch TV
Mom's Pearl Necklace Jewelery       30 Inch Pearl Neclace
```

And to see specific electronics items with their information, you can use a query such as this, with an inner join to the parent table to get the basic nonspecific information:

```
SELECT  Item.Name, ElectronicItem.BrandName, ElectronicItem.ModelNumber, ElectronicItem.SerialNumber
FROM    Inventory.ElectronicItem
            JOIN Inventory.Item
                  ON Item.ItemId = ElectronicItem.ItemId;
```

This returns

```
Name              BrandName     ModelNumber       SerialNumber
----------------  ---------     -----------       ------------
40 Inch Sorny TV  Sorny         R2D12             XD49393
29 Inch JQC TV    JVC           CRTVX29           NULL
```

Finally, it is also quite common that you may want to see a complete inventory with the specific information, since this is truly the natural way you think of the data and why the typical designer will design the table in a single table no matter what. We return an extended description column this time by formatting the data based on the type of row:

```
SELECT Name, Description,
       CASE Type
         WHEN 'Electronics'
           THEN 'Brand:' + COALESCE(BrandName,'_______') +
             ' Model:' + COALESCE(ModelNumber,'________') +
             ' SerialNumber:' + COALESCE(SerialNumber,'_______')
```


www.it-ebooks.info

```
        WHEN 'Jewelery'
          THEN 'QualityLevel:' + QualityLevel +
            ' Appraiser:' + COALESCE(AppraiserName,'_______') +
            ' AppraisalValue:' +COALESCE(Cast(AppraisalValue as varchar(20)),'_______')
           +' AppraisalYear:' + COALESCE(AppraisalYear,'____')
          ELSE '' END as ExtendedDescription
FROM Inventory.Item --outer joins because every item will only have one of these if any
      LEFT OUTER JOIN Inventory.ElectronicItem
        ON Item.ItemId = ElectronicItem.ItemId
      LEFT OUTER JOIN Inventory.JeweleryItem
        ON Item.ItemId = JeweleryItem.ItemId;
```

This returns a formatted description:

```
Name                 Description              ExtendedDescription
-------------------  -------------------------  -------------------------------------------
Den Couch            Blue plaid couch, seats 4
Den Ottoman          Blue plaid ottoman that …
40 Inch Sorny TV     40 Inch TV               Brand:Sorny Model:R2D12 SerialNumber:XD49393
29 Inch JQC TV       29 Inch TV               Brand:JVC Model:CRTVX29 SerialNumber:_______
Mom's Pearl Necklace 30 Inch Pearl Neclace    NULL
```

The point of this section on generalization really goes back to the basic precepts that you design for the user's needs. If we had created a table per type of item in the house: `Inventory.Lamp`, `Inventory.ClothesHanger`, and so on, the process of normalization gets the blame. But the truth is, if you really listen to the user's needs and model them correctly, you may never need to consider generalizing your objects. However, it is still a good thing to consider, looking for commonality among the objects in your database looking for cases where you could get away with less tables rather than more.

■ **Tip** It may seem unnecessary, even for a simple home inventory system to take these extra steps in your design. However, the point I am trying to make here is that if you have rules about how data should look, almost certainly having a column for that data is going to make more sense. Even if your business rule enforcement is as minimal as just using the final query, it will be far more obvious to the end user that the `SerialNumber: _____` value is a missing value that probably needs to be filled in.

Storing User-Specified Data

One of the common problems that has no comfortable solution is giving users a method to expand the catalog of values they can store without losing control of the database design. The biggest issue is the integrity of the data that they want to store in this database, in that it is very rare that you want to store data and not use it to make decisions. In this section, I will explore a couple of the common methods for doing expanding the data catalog by the end user.

As I have tried to make clear throughout the rest of the book so far, relational tables are not meant to be flexible. SQL Server code is not really meant to be overly flexible. T-SQL as a language is not made for flexibility

www.it-ebooks.info

(at least not from the standpoint of producing reliable databases that produce expected results and producing good performance while protecting the quality of the data, which I have said many times is almost always the most important thing).

Unfortunately, reality is that users want flexibility, and frankly, you can't tell users that they can't get what they want, when they want it, and in the form they want it. As an architect, I want to give the users what they want, within the confines of reality and sensibility, so it is necessary to ascertain some method of giving the user the flexibility they demand, along with methods to deal with this data in a manner that feels good to them. The technique used to solve this problem is pretty simple. It requires the design to be flexible enough to morph to the needs of the user, without the intervention of a database programmer manually changing the catalog. But with all problems, there are generally a couple of solutions that can be used to improve them.

■ **Note** I will specifically speak only of methods that are methods that allow you to work with the relational engine in a seminatural manner. There are others, such using an XML column that can be used, but they require you to use completely different methods for dealing with the data.

The methods I will demonstrate are as follows:

- Big old list of generic columns
- Entity-attribute-value (EAV)
- Adding columns to the table, likely using sparse columns

The last time I had this type of need was to gather the properties on networking equipment. Each router, modem, and so on for a network has various properties (and hundreds or thousands of them at that). For this section, I will present this example as three different examples.

The basis of this example will be a simple table called `Equipment`. It will have a surrogate key and a tag that will identify it. It is created using the following code:

```
CREATE SCHEMA Hardware;
GO
CREATE TABLE Hardware.Equipment
(
  EquipmentId int NOT NULL
        CONSTRAINT PKHardwareEquipment PRIMARY KEY,
  EquipmentTag varchar(10) NOT NULL
        CONSTRAINT AKHardwareEquipment UNIQUE,
  EquipmentType varchar(10)
);
GO
INSERT  INTO Hardware.Equipment
VALUES  (1,'CLAWHAMMER','Hammer'),
        (2,'HANDSAW','Saw'),
        (3,'POWERDRILL','PowerTool');
```

By this point in this book, you should know that this is not how the whole table would look in the actual solutions, but these three columns will give you enough to build an example from.

www.it-ebooks.info

Big Old List of Generic Columns

The basics of this plan is to create your normalized tables, make sure you have everything nicely normalized, and then, just in case the user wants to store "additional" information, create a bunch of generic columns. This sort of tactic is quite common in product-offering databases, where the people who are building the database are trying to make sure that the end users have some space in the tool for their custom application.

For our example, adjusting the `Equipment` table, someone might implement a set of columns in their table such as the following alternate version of our original table, shown in Figure 8-16.

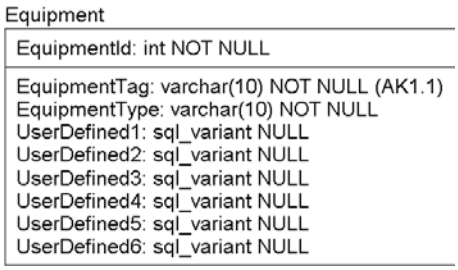


Figure 8-16. *Equipment table with UserDefined columns*

On the good hand, the end user has a place to store some values that they might need. On the not so good hand, without some very stringent corporate policy, these columns will just be used in any manner that each user decides. This is often the case, which leads to chaos when using this data in SQL. In the best case, `userDefined1` is where users always put the `schmeglin` number, and the second the `cooflin` diameter or some other value. If the programmer building the user interface is "clever," he could give names to these columns in some metadata that he defines so that they show up on the screen with usable names.

The next problem, though, comes for the person who is trying to query the data, because knowing what each column means is too often a mystery. If you are lucky, there will be metadata you can query and then build views, and so on. The problem is that you have put the impetus on the end user to know this before using the data, whereas in the other methods, the definition will be part of even creating storage points, and the usage of them will be slightly more natural, if sometimes a bit more troublesome.

From a physical implementation/tuning point of view, the problem with this is that by adding these `sql_variant` columns, you are potentially bloating your base table, making any scans of the table take longer, increasing page splits, and so on. I won't produce sample code for this because it would be my goal to dissuade you from using this pattern of implementing user-defined data, but it is a technique that is commonly done.

Entity-Attribute-Value (EAV)

The next method of implementing user-specified data is using the entity-attribute-value (EAV) method. These are also known by a few different names, such as property tables, loose schemas, or open schema. In 2005 and earlier, this technique was generally considered the default method of implementing a table to allow users to configure their own storage and is still a commonly used pattern.

The basic idea behind this method is to have another related table associated with the table you want to add information about. This table will hold the values that you want to store. Then, you can either include the name of the attribute in the property table or (as I will do) have a table that defines the basic properties of a property.

Considering our needs with equipment, I will use the model shown in Figure 8-17.

www.it-ebooks.info

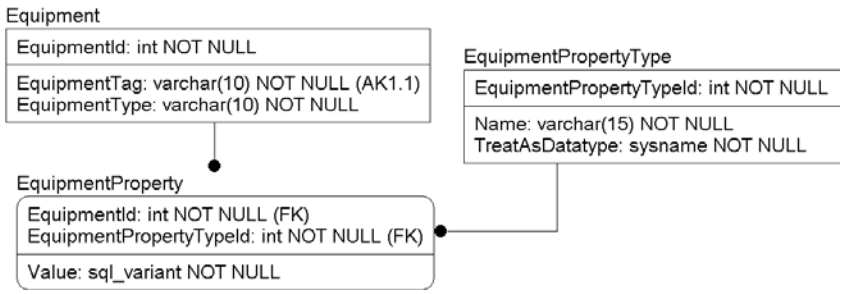


Figure 8-17. *Property schema for storing equipment properties with unknown attributes*

Using these values in queries isn't a natural task at all, and as such, you should avoid loose schemas like this unless absolutely necessary. The main reason is that you should rarely, if ever, have a limit to the types of data that the user can store. For example, if you as the architect know that you want to allow only three types of properties, you should almost never use this technique because it is almost certainly better to add the three known columns, possibly using the techniques for subtyped entities presented earlier in the book to implement the different tables to hold the values that pertain to only one type or another. The goal here is to build loose objects that can be expanded for some reason or another. In our example, it is possible that the people who develop the equipment you are working with will add a property that you want to then keep up with. In my real-life usage of this technique, there were hundreds of properties added as different equipment was brought online, and each device was interrogated for its properties.

To do this, I will create an `EquipmentPropertyType` table and add a few types of properties:

```
CREATE TABLE Hardware.EquipmentPropertyType
(
        EquipmentPropertyTypeId int NOT NULL
                CONSTRAINT PKHardwareEquipmentPropertyType PRIMARY KEY,
        Name varchar(15) NOT NULL
                CONSTRAINT AKHardwareEquipmentPropertyType UNIQUE,
        TreatAsDatatype sysname NOT NULL
);

INSERT INTO Hardware.EquipmentPropertyType
VALUES  (1,'Width','numeric(10,2)'),
        (2,'Length','numeric(10,2)'),
        (3,'HammerHeadStyle','varchar(30)');
```

Then, I create an `EquipmentProperty` table: that will hold the actual property values. I will use a `sql_variant` type for the value column to allow any type of data to be stored, but it is also typical to either use a character string type value (requiring the caller/user to convert to a string representation of all values) or having multiple columns, on of each possible type. Both of these and using `sql_variant` all have slight difficulties, but I tend to use `sql_variant` because the data is stored in its native format and is usable in many ways in its current format. In the definition of the property, I will also include the datatype that I expect the data to be, and in our insert procedure, we will test the data to make sure it meets the requirements for a specific datatype.

www.it-ebooks.info

```
CREATE TABLE Hardware.EquipmentProperty
(
        EquipmentId int NOT NULL
                CONSTRAINT FKHardwareEquipment$hasExtendedPropertiesIn$HardwareEquipmentProperty
                        REFERENCES Hardware.Equipment(EquipmentId),
        EquipmentPropertyTypeId int NOT NULL
             CONSTRAINT FKHardwareEquipmentPropertyTypeId$definesTypesFor$HardwareEquipmentProperty
                        REFERENCES Hardware.EquipmentPropertyType(EquipmentPropertyTypeId),
        Value sql_variant NULL,
        CONSTRAINT PKHardwareEquipmentProperty PRIMARY KEY
                (EquipmentId, EquipmentPropertyTypeId)
);
```

Then, I need to load some data. For this task, I will build a procedure that can be used to insert the data by name and, at the same time, will validate that the datatype is right. That is a bit tricky because of the `sql_variant` type, and it is one reason that property tables are sometimes built using character values. Since everything has a textual representation and it is easier to work with in code, it just makes things simpler for the code but often far worse for the storage engine to maintain.

In the procedure, I will insert the row into the table and then use dynamic SQL to validate the value by casting the value as the datatype the user passed in. (Note that the procedure follows the standards that I will establish in later chapters for transactions and error handling. I don't always do this in this chapter to keep the samples clean, but this procedure deals with validations.)

```
CREATE PROCEDURE Hardware.EquipmentProperty$Insert
(
        @EquipmentId int,
        @EquipmentPropertyName varchar(15),
        @Value sql_variant
)
AS
        SET NOCOUNT ON;
        DECLARE @entryTrancount int = @@trancount;
        BEGIN TRY
          DECLARE @EquipmentPropertyTypeId int,
                  @TreatAsDatatype sysname;

          SELECT  @TreatAsDatatype = TreatAsDatatype,
                  @EquipmentPropertyTypeId = EquipmentPropertyTypeId
          FROM    Hardware.EquipmentPropertyType
          WHERE   EquipmentPropertyType.Name = @EquipmentPropertyName;

        BEGIN TRANSACTION;
          --insert the value
          INSERT INTO Hardware.EquipmentProperty(EquipmentId, EquipmentPropertyTypeId,
                Value)
          VALUES (@EquipmentId, @EquipmentPropertyTypeId, @Value);

          --Then get that value from the table and cast it in a dynamic SQL
          -- call. This will raise a trappable error if the type is incompatible
          DECLARE @validationQuery varchar(max) =
                ' DECLARE @value sql_variant
                  SELECT @value = cast(value AS ' + @TreatAsDatatype + ')
                  FROM Hardware.EquipmentProperty
```


www.it-ebooks.info

```
                    WHERE EquipmentId = ' + CAST (@EquipmentId AS varchar(10)) + '

                          and EquipmentPropertyTypeId = ' +
                            cast(@EquipmentPropertyTypeId AS varchar(10)) + ' ';

            EXECUTE (@validationQuery);
         COMMIT TRANSACTION;
      END TRY
      BEGIN CATCH
         IF @@TRANCOUNT > 0
           ROLLBACK TRANSACTION;

         DECLARE @ERRORmessage nvarchar(4000)
         SET @ERRORmessage = 'Error occurred in procedure ''' +
                 object_name(@@procid) + ''', Original Message: '''
                 + ERROR_MESSAGE() + '''';
         THROW 50000,@ERRORMessage,16;
         RETURN -100;

      END CATCH
```

So, if you try to put in an invalid piece of data:

```
EXEC Hardware.EquipmentProperty$Insert 1,'Width','Claw'; --width is numeric(10,2)
```

you will get the following error:

```
Msg 50000, Level 16, State 16, Procedure EquipmentProperty$Insert, Line 48
Error occurred in procedure 'EquipmentProperty$Insert', Original Message: 'Error converting
data type varchar to numeric.'
```

Now, I create some proper demonstration data:

```
EXEC Hardware.EquipmentProperty$Insert @EquipmentId =1,
        @EquipmentPropertyName = 'Width', @Value = 2;
EXEC Hardware.EquipmentProperty$Insert @EquipmentId =1,
        @EquipmentPropertyName = 'Length',@Value = 8.4;
EXEC Hardware.EquipmentProperty$Insert @EquipmentId =1,
        @EquipmentPropertyName = 'HammerHeadStyle',@Value = 'Claw'
EXEC Hardware.EquipmentProperty$Insert @EquipmentId =2,
        @EquipmentPropertyName = 'Width',@Value = 1;
EXEC Hardware.EquipmentProperty$Insert @EquipmentId =2,
        @EquipmentPropertyName = 'Length',@Value = 7;
EXEC Hardware.EquipmentProperty$Insert @EquipmentId =3,
        @EquipmentPropertyName = 'Width',@Value = 6;
EXEC Hardware.EquipmentProperty$Insert @EquipmentId =3,
        @EquipmentPropertyName = 'Length',@Value = 12.1;
```

To view the data in a raw manner, I can simply query the data as such:

```
SELECT  Equipment.EquipmentTag,Equipment.EquipmentType,
        EquipmentPropertyType.name, EquipmentProperty.Value
```


www.it-ebooks.info

```
FROM    Hardware.EquipmentProperty
          JOIN Hardware.Equipment
                ON Equipment.EquipmentId = EquipmentProperty.EquipmentId
          JOIN Hardware.EquipmentPropertyType
                ON EquipmentPropertyType.EquipmentPropertyTypeId =
                        EquipmentProperty.EquipmentPropertyTypeId;
```

This is usable but not very natural as results:

```
EquipmentTag      EquipmentType       name               Value
------------      -------------       ---------------    -----
CLAWHAMMER        Hammer              Width              2
CLAWHAMMER        Hammer              Length             8.4
CLAWHAMMER        Hammer              HammerHeadStyle    Claw
HANDSAW           Saw                 Width              1
HANDSAW           Saw                 Length             7
POWERDRILL        PowerTool           Width              6
POWERDRILL        PowerTool           Length             12.1
```

To view this in a natural, tabular format along with the other columns of the table, I could use `PIVOT`, but the "old" style method to perform a pivot, using `MAX()` aggregates, works better here because I can fairly easily make the statement dynamic (which is the next query sample):

```
SET ANSI_WARNINGS OFF; --eliminates the NULL warning on aggregates.
SELECT Equipment.EquipmentTag,Equipment.EquipmentType,
  MAX(CASE WHEN EquipmentPropertyType.name = 'HammerHeadStyle' THEN Value END)
        AS 'HammerHeadStyle',
  MAX(CASE WHEN EquipmentPropertyType.name = 'Length'THEN Value END) AS Length,
  MAX(CASE WHEN EquipmentPropertyType.name = 'Width' THEN Value END) AS Width
FROM Hardware.EquipmentProperty
        JOIN Hardware.Equipment
          ON Equipment.EquipmentId = EquipmentProperty.EquipmentId
        JOIN Hardware.EquipmentPropertyType
          ON EquipmentPropertyType.EquipmentPropertyTypeId =
             EquipmentProperty.EquipmentPropertyTypeId
GROUP BY Equipment.EquipmentTag,Equipment.EquipmentType;
SET ANSI_WARNINGS OFF; --eliminates the NULL warning on aggregates.
```

This returns the following:

```
EquipmentTag      EquipmentType      HammerHeadStyle      Length      Width
------------      -------------      ---------------      ------      -----
CLAWHAMMER        Hammer             Claw                 8.40        2.00
HANDSAW           Saw                NULL                 7.00        1.00
POWERDRILL        PowerTool          NULL                 12.10       6.00
```

If you execute this on your own in the text mode, what you will quickly notice is how much editing I had to do to the data. Each `sql_variant` column will be formatted for a huge amount of data. And, you had to manually set up the values. In the following extension, I have used `XML PATH` to output the different properties to different

www.it-ebooks.info

columns, starting with MAX. (This is a common SQL Server 2005 and later technique for turning rows into columns. Do a search for turning rows into columns in SQL Server, and you will find the details.)

```
SET ANSI_WARNINGS OFF;
DECLARE @query varchar(8000);
SELECT  @query = 'SELECT Equipment.EquipmentTag,Equipment.EquipmentType ' + (
                 SELECT DISTINCT
                   ',MAX(CASE WHEN EquipmentPropertyType.name = ''' +
                         EquipmentPropertyType.name + ''' THEN CAST(Value AS ' +
                         EquipmentPropertyType.TreatAsDatatype + ') END) AS [' +
                         EquipmentPropertyType.name + ']' AS [text()]
                 FROM
                   Hardware.EquipmentPropertyType
                 FOR XML PATH('') ) + '
                 FROM Hardware.EquipmentProperty
                         JOIN Hardware.Equipment
                           ON Equipment.EquipmentId =
                             EquipmentProperty.EquipmentId
                         JOIN Hardware.EquipmentPropertyType
                           ON EquipmentPropertyType.EquipmentPropertyTypeId
                             = EquipmentProperty.EquipmentPropertyTypeId
        GROUP BY Equipment.EquipmentTag,Equipment.EquipmentType '
EXEC (@query);
```

Executing this will get you the following:

```
EquipmentTag    EquipmentType    HammerHeadStyle    Length    Width
------------    -------------    ---------------    ------    -----
CLAWHAMMER      Hammer           Claw               8.40      2.00
HANDSAW         Saw              NULL               7.00      1.00
POWERDRILL      PowerTool        NULL               12.10     6.00
```

I won't pretend that I didn't have to edit the results to get them to fit, but each of these columns was formatted as the datatype that you specified in the EquipmentPropertyType table, not as 8,000-character values (that is a lot of little minus signs under each heading).

One thing I won't go any further into in this example is the EquipmentType column and how you might use it to limit certain properties to apply only to certain types of equipment. It would require adding a table for type and relating it to the Equipment and EquipmentPropertyType tables. Then, you could build even smarter display procedures by asking only for columns of type HandTool. Then, the display routine would get back only those properties that are for the type you want.

■ **Tip** The query that was generated to create the output in "relational" manner can easily be turned into a view for permanent usage. You can even create such a view instead of triggers to make the view treat the data like relational data. All of this could be done by your toolset as well, if you really do need to use the EAV pattern to stored data. In the next section, I will propose an alternative method that is a bit easier to work with, but there are many people who swear by this method.

---

www.it-ebooks.info

Adding Columns to a Table

For the final choice that I will demonstrate, consider the idea of using the facilities that SQL Server gives us for implementing columns, rather than implementing your own metadata system. In the previous examples, it was impossible to use the table structures in a natural way, meaning that if you wanted to query the data, you had to know what was meant by interrogating the metadata. In the EAV solution, a normal `SELECT` statement was almost impossible. One could be simulated with a dynamic stored procedure, or you could possibly create a hard-coded view, but it certainly would not be easy for the typical end user without the aid of a programmer.

Note that the columns you create do not need to be on the base tables. You could easily build a separate table, just like the EAV solution in the previous section, but instead of building your own method of storing data, add columns to the new property table. Using the primary key of the existing table to implement it as a one- to zero-on-one cardinality relationship will keep users from needing to modify the main table.

■ **Tip** Always have a way to validate the schema of your database. If this is a corporate situation, a simple copy of the database structure might be good enough. If you ship a product to a customer, you should produce an application to validate the structures against before applying a patch or upgrade or even allowing your tech support to help out with a problem. Although you cannot stop a customer from making a change (like a new column, index, trigger, or whatever), you don't want the change to cause an issue that your tech support won't immediately recognize.

The key to this method is to use SQL Server more or less naturally (there may still be some metadata required to manage data rules, but it is possible to use native SQL commands with the data). Instead of all the stuff we went through in the previous section to save and view the data, just use `ALTER TABLE`, and add the column. Of course, it isn't necessarily as easy as making changes to the tables and granting control of the tables to the user, especially if you are going to allow non-administrative users to add their own columns ad hoc. However, building a stored procedure or two would allow the user to add columns to the table (and possibly remove them from the table), but you might want to allow only those columns with a certain prefix to be added, or you could use extended properties. This would just be to prevent "whoops" events from occurring, not to prevent an administrator from dropping a column. You really can't prevent an administrative user of an instance from dropping anything in the database unless you lock things down far too much from your customers. If the application doesn't work because they have removed columns from the application that are necessary, well, that is going to be the DBA's fault.

A possible downside of using this method on the base table is that you can really affect the performance of a system if you just allow users to randomly add large amounts of data to your tables. In some cases, the user who is extending the database will have a need for a column that is very important to their usage of the system, like a key value to an external system. For those needs, you might want to add a column to the base table. But if the user wants to store just a piece of data on a few rows, it may not make any sense to add this column to the base physical table (more on the physical aspects of storing data in Chapter 10), especially if it will not be used on many rows.

In versions prior to SQL Server 2008, to use this method, I would have likely built another table to hold these additional columns and then joined to the table to fetch the values or possibly built an XML column.

In SQL Server 2008, a method was added that gave use the best of both solutions with sparse columns. A *sparse column* is a type of column storage where a column that is `NULL` takes no storage at all (normal `NULL` columns require space to indicate that they are `NULL`). Basically, the data is stored internally as a form of an EAV\XML solution that is associated with each row in the table. Sparse columns are added and dropped from the table using the same DDL statements as normal columns (with the added keyword of `SPARSE` on the column create statement). You can also use the same DML operations on the data as you can for regular tables. However, since the purpose of having sparse columns is to allow you to add many columns to the table (the maximum is 30,000!),

www.it-ebooks.info

you can also work with sparse columns using a *column set*, which gives you the ability to retrieve and work with only the sparse columns that you desire to or that have values in the row.

Sparse columns are slightly less efficient in many ways when compared to normal columns, so the idea would be to add nonsparse columns to your tables when they will be used quite often, and if they will pertain only to rare or certain types of rows, then you could use a sparse column. Several types cannot be stored as sparse. These are as follows:

- The spatial types
- `timestamp`
- User-defined datatypes
- `text`, `ntext`, and `image` (Note that you shouldn't use these anyway; use `varchar(max)`, `nvarchar(max)`, and `varbinary(max)` instead.)

Returning to the `Equipment` example, all I'm going to use this time is the single table. Note that the data I want to produce looks like this:

```
EquipmentTag       EquipmentType      HammerHeadStyle      Length      Width
-----------        -------------      ---------------      ------      -----
CLAWHAMMER         Hammer             Claw                 8.40        2.00
HANDSAW            Saw                NULL                 7.00        1.00
POWERDRILL         PowerTool          NULL                 12.10       6.00
```

To add the `Length` column to the `Equipment` table, use this:

```
ALTER TABLE Hardware.Equipment
        ADD Length numeric(10,2) SPARSE NULL;
```

If you were building an application to add a column, you could use a procedure like the following to give the user rights to add a column without getting all the other control types over the table. Note that if you are going to allow users to drop columns, you will want to use some mechanism to prevent them from dropping primary system columns, such as a naming standard or extended property. You also may want to employ some manner of control to prevent them from doing this at just any time they want.

```
CREATE PROCEDURE Hardware.Equipment$addProperty
(
        @propertyName sysname, --the column to add
        @datatype     sysname, --the datatype as it appears in a column creation
        @sparselyPopulatedFlag bit = 1 --Add column as sparse or not
)
WITH EXECUTE AS SELF
AS
  --note: I did not include full error handling for clarity
  DECLARE @query nvarchar(max);

  --check for column existance
```


www.it-ebooks.info

```
  IF NOT EXISTS (SELECT *
                 FROM   sys.columns
                 WHERE  name = @propertyName
                  AND   OBJECT_NAME(object_id) = 'Equipment'
                  AND   OBJECT_SCHEMA_NAME(object_id) = 'Hardware')
  BEGIN
    --build the ALTER statement, then execute it
    SET @query = 'ALTER TABLE Hardware.Equipment ADD ' + quotename(@propertyName) + ' '
            + @datatype
            + case when @sparselyPopulatedFlag = 1 then ' SPARSE ' end
            + ' NULL ';
    EXEC (@query);
  END
ELSE
    THROW 50000, 'The property you are adding already exists',16;
```

Now, any user you give rights to run this procedure can add a column to the table:

```
--EXEC  Hardware.Equipment$addProperty 'Length','numeric(10,2)',1; -- added manually
EXEC    Hardware.Equipment$addProperty 'Width','numeric(10,2)',1;
EXEC    Hardware.Equipment$addProperty 'HammerHeadStyle','varchar(30)',1;
```

Viewing the table, you see the following:

```
SELECT  EquipmentTag, EquipmentType, HammerHeadStyle,Length,Width
FROM    Hardware.Equipment;
```

which returns the following (I will use this SELECT statement several times):

```
EquipmentTag    EquipmentType    HammerHeadStyle     Length    Width
------------    -------------    ---------------     ------    -----
CLAWHAMMER      Hammer           NULL                NULL      NULL
HANDSAW         Saw              NULL                NULL      NULL
POWERDRILL      PowerTool        NULL                NULL      NULL
```

Now, you can treat the new columns just like they were normal columns. You can update them using a normal UPDATE statement:

```
UPDATE  Hardware.Equipment
SET     Length = 7.00,
        Width = 1.00
WHERE   EquipmentTag = 'HANDSAW';
```

Checking the data, you can see that the data was updated:

```
EquipmentTag    EquipmentType    HammerHeadStyle     Length    Width
------------    -------------    ---------------     ------    -----
CLAWHAMMER      Hammer           NULL                NULL      NULL
HANDSAW         Saw              NULL                7.00      1.00
POWERDRILL      PowerTool        NULL                NULL      NULL
```

One thing that is so much easier using this method of user-specified columns is validation. Because the columns behave just like columns should, you can use a CHECK constraint to validate row-based constraints:

www.it-ebooks.info

```
ALTER TABLE Hardware.Equipment
    ADD CONSTRAINT CHKHardwareEquipment$HammerHeadStyle CHECK
        ((HammerHeadStyle is NULL AND EquipmentType <> 'Hammer')
        OR EquipmentType = 'Hammer');
```

■ **Note** You could easily create a procedure to manage a user-defined check constraint on the data just like I created the columns.

Now, if you try to set an invalid value, like a saw with a HammerHeadStyle, you get an error:

```
UPDATE  Hardware.Equipment
SET     Length = 12.10,
        Width = 6.00,
        HammerHeadStyle = 'Wrong!'
WHERE   EquipmentTag = 'HANDSAW';
```

This returns the following:

```
Msg 547, Level 16, State 0, Line 1
The UPDATE statement conflicted with the CHECK constraint "CHKHardwareEquipment$HammerHeadStyle".
The conflict occurred in database "Chapter8", table "Hardware.Equipment".
```

Setting the rest of the values, I return to where I was in the previous section's data, only this time the SELECT statement could have been written by a novice:

```
UPDATE  Hardware.Equipment
SET     Length = 12.10,
        Width = 6.00
WHERE   EquipmentTag = 'POWERDRILL';

UPDATE  Hardware.Equipment
SET     Length = 8.40,
        Width = 2.00,
        HammerHeadStyle = 'Claw'
WHERE   EquipmentTag = 'CLAWHAMMER';

GO
SELECT EquipmentTag, EquipmentType, HammerHeadStyle ,Length,Width
FROM   Hardware.Equipment;
```

which returns that result set I was shooting for:

```
EquipmentTag      EquipmentType      HammerHeadStyle     Length      Width
------------      -------------      ---------------     ------      -----
CLAWHAMMER        Hammer             Claw                8.40        2.00
HANDSAW           Saw                NULL                7.00        1.00
POWERDRILL        PowerTool          NULL                12.10       6.00
```


www.it-ebooks.info

Now, up to this point, it really did not make any difference if this was a SPARSE column or not. Even if I just used a SELECT * from the table, it would look just like a normal set of data. Pretty much the only way you can tell is by looking at the metadata:

```
SELECT  name, is_sparse
FROM    sys.columns
WHERE   OBJECT_NAME(object_id) = 'Equipment'
```

This returns the following:

```
name                is_sparse
---------------     ---------
EquipmentId         0
EquipmentTag        0
EquipmentType       0
Length              1
Width               1
HammerHeadStyle     1
```

There is a different way of working with this data that can be much easier to deal with if you have many sparse columns with only a few of them filled in. You can define a column that defines a *column set*, which is the XML representation of the set of columns that are stored for the sparse column. With a column set defined, you can access the XML that manages the sparse columns and work with it directly. This is handy for dealing with tables that have a lot of empty sparse columns, because NULL sparse columns do not show up in the XML, allowing you to pass very small amounts of data to the user interface, though it will have to deal with it as XML rather than in a tabular data stream.

■ **Tip** You cannot add or drop the column set once there are sparse columns in the table, so decide which to use carefully.

For our table, I will drop the check constraint and sparse columns and add a column set (you cannot modify the column set when any sparse columns exist, presumably because this is something new that they have not added yet in 2008, and in addition, you may have only one):

```
ALTER TABLE Hardware.Equipment
        DROP CONSTRAINT CHKHardwareEquipment$HammerHeadStyle;
ALTER TABLE Hardware.Equipment
        DROP COLUMN HammerHeadStyle, Length, Width;
```

Now, I add a column set, which I will name SparseColumns:

```
ALTER TABLE Hardware.Equipment
      ADD SparseColumns xml column_set FOR ALL_SPARSE_COLUMNS;
```

Now, I add back the sparse columns and constraints using my existing procedure:

```
EXEC Hardware.equipment$addProperty 'Length','numeric(10,2)',1;
```


www.it-ebooks.info

```
EXEC Hardware.equipment$addProperty 'Width','numeric(10,2)',1;
EXEC Hardware.equipment$addProperty 'HammerHeadStyle','varchar(30)',1;
GO
ALTER TABLE Hardware.Equipment
    ADD CONSTRAINT CHKHardwareEquipment$HammerHeadStyle CHECK
        ((HammerHeadStyle is NULL AND EquipmentType <> 'Hammer')
        OR EquipmentType = 'Hammer');
```

Now, I can still update the columns individually using the UPDATE statement:

```
UPDATE  Hardware.Equipment
SET     Length = 7,
        Width = 1
WHERE   EquipmentTag = 'HANDSAW';
```

But this time, using SELECT * does not return the sparse columns as normal SQL columns; it returns them as XML:

```
SELECT  *
FROM    Hardware.Equipment;
```

This returns the following:

```
EquipmentId     EquipmentTag      EquipmentType      SparseColumns
-----------     ------------      -------------      ----------------------------------------
1               CLAWHAMMER        Hammer             NULL
2               HANDSAW           Saw                <Length>7.00</Length><Width>1.00</Width>
3               POWERDRILL        PowerTool          NULL
```

You can also update the SparseColumns column directly using the XML representation:

```
UPDATE  Hardware.Equipment
SET     SparseColumns = '<Length>12.10</Length><Width>6.00</Width>'
WHERE   EquipmentTag = 'POWERDRILL';

UPDATE  Hardware.Equipment
SET     SparseColumns = '<Length>8.40</Length><Width>2.00</Width>
                         <HammerHeadStyle>Claw</HammerHeadStyle>'
WHERE   EquipmentTag = 'CLAWHAMMER';
```

Enumerating the columns gives us the output that matches what we expect:

```
SELECT  EquipmentTag, EquipmentType, HammerHeadStyle, Length, Width
FROM    Hardware.Equipment;
```

Finally, we're back to the same results as before:

```
EquipmentTag    EquipmentType      HammerHeadStyle    Length      Width
------------    -------------      ---------------    ------      -----
CLAWHAMMER      Hammer             Claw               8.40        2.00
HANDSAW         Saw                NULL               7.00        1.00
POWERDRILL      PowerTool          NULL               12.10       6.00
```


www.it-ebooks.info

Sparse columns can be indexed, but you will likely want to create a filtered index (discussed earlier in this chapter for selective uniqueness). The `WHERE` clause of the filtered index could be used either to associate the index with the type of row that makes sense (like in our `HAMMER` example's `CHECK` constraint, you would likely want to include `EquipmentTag` and `HammerHeadStyle`) or to simply ignore `NULL`.

In comparison to the methods used with property tables, this method is going to be tremendously easier to implement, and if you are able to use sparse columns, it's faster and far more natural to work with in comparison to the EAV method. It is going to feel strange allowing users to change the table structures of your main data tables, but with proper coding, testing, and security practices (and perhaps a DDL trigger monitoring your structures for changes to let you know when these columns are added), you will end up with a far better-performing and more flexible system.

Anti-Patterns

In many ways, you, as a reader so far in the book, probably think that I worship Codd and all of his rules with such reverence that I would get together an army of vacuuming robots and start a crusade in his name (and if you are still looking for pictures of hot "models" in this book, you now are probably concerned that this is a secret fish worship book), and in a way, you would be correct (other than the fish worship thing, of course). Codd was the theorist who got the relational model rolling and put the theories into action for years and years. It has to be noted that his theories have held up for 30 years now and are just now being realized in full, as hardware is getting more and more powerful. His goals of an invisible physical layer for the relational user is getting closer and closer, though we still need physical understanding of the data for performance tuning purposes, which is why I reluctantly included Chapter 10 on physical database structures.

That having been said, I am equally open to new ideas. Things like identity columns are offensive to many purists, and I am very much a big fan of them. For every good idea that comes out to challenge solid theory, there come many that fail to work (can't blame folks for trying). In this section, I will outline four of these practices and explain why I think they are such bad ideas:

- *Undecipherable data*: Too often, you find the value 1 in a column with no idea what "1" means without looking into copious amounts of code.
- *One-size-fits-all domain*: One domain table is used to implement all domains rather than using individual tables that are smaller and more precise.
- *Generic key references*: In this anti-pattern, you have one column where the data in the column might be the key from any number of tables, requiring you to decode the value rather than know what it is.
- *Overusing unstructured data*: This is the bane of existence for DBAs—the blob of text column that the users swear they put well-structured data in for you to parse out. You can't eliminate a column for notes here and there, but overuse of such constructs lead to lots of DBA pain.

There are a few other problematic patterns I need to reiterate (with chapter references), in case you have read only this chapter so far. My goal in this section is to hit upon some patterns that would not come up in the "right" manner of designing a database but are common ideas that designers get when they haven't gone through the heartache of these patterns:

www.it-ebooks.info

- *Poor normalization practices*: Normalization is an essential part of the process of database design, and it is far easier to achieve than it will seem when you first start. And don't be fooled by people who say that Third Normal Form is the ultimate level; Fourth Normal Form is very important and common as well. (Chapter 5 covers normalization in depth.)
- *Poor domain choices*: Lots of databases just use `varchar(50)` for every nonkey column, rather than taking the time to determine proper domains for their data. Sometimes, this is even true of columns that are related via foreign key and primary key columns, which makes the optimizer work harder. See Chapter 5.
- *No standardization of datatypes*: It is a good idea to make sure you use the same sized/typed column whenever you encounter like typed things. For example, if your company's account number is `char(9)`, just don't have it 20 different ways: `varchar(10)`, `varchar(20)`, `char(15)`, and so on. All of these will store the data losslessly, but only `char(9)` will be best and will help keep your users from needing to think about how to deal with the data. (See Chapter 6 for more discussion of choosing a datatype and Appendix A for a more detailed list and discussion of all of the intrinsic relational types.)

And yes, there are many more things you probably shouldn't do, but this section has listed some of the bigger design-oriented issues that really drive you crazy when you have to deal with the aftermath of their use.

The most important issue to understand (if *Star Trek* has taught us anything) is that if you use one of these anti-patterns along with the other patterns discussed in this chapter, the result will likely be mutual annihilation.

Undecipherable Data

One of the most annoying things when dealing with a database designed by a typical programmer is undecipherable values. Code such as `WHERE status = 1` will pepper the code you no doubt discover using profiler, and you as the data developer end up scratching your head in wonderment as to what 5 represents. Of course, the reason for this is that the developers don't think of the database as a primary data resource, rather they think of the database as simply the place where they hold state for their objects.

Of course, in their code, they are probably doing a decent job of presenting the meaning of the values in their coding. They aren't actually dealing with a bunch of numbers in their code; they have a constant structure, such as

```
CONST (CONST_Active = 1, CONST_Inactive = 0);
```

So the code they are using to generate the code make sense because they have said `"WHERE status = " & CONST_Active`. This is clear in the usage but not clear at the database level (where the values are actually seen and used by everyone else!). From a database standpoint, we have a few possibilities:

- Use descriptive values such as "Active" and "Inactive" directly. This makes the data more decipherable but doesn't provide a domain of possible values. If you have no inactive values, you will not know about its existence at the database
- Create tables to implement a domain. Have a table with all possible values.

For the latter, your table could use the descriptive values as the domain, or you can use the integer values that the programmer likes as well. Yes, there will be double definitions of the values (one in the table, one in the constant declaration), but since domains such as this rarely change, it is generally not a terrible issue. The principles I tend to try to design by follow:

www.it-ebooks.info

- Only have values that can be deciphered using the database:
  - Foreign key to a lookup table
  - Human readable values with no expansion in CASE expressions.
  - *No bitmasks*! (We are not writing machine code!)
- Don't be afraid to have lots of small tables. Joins generally cost a lot less than the time needed to decipher a value, measured in programmer time, ETL time, and end user frustration.

One-Size-Fits-All Key Domain

Relational databases are based on the fundamental idea that every object represents one and only one thing. There should never be any doubt as to what a piece of data refers. By tracing through the relationships, from column name to table name to primary key, it should be easy to examine the relationships and know exactly what a piece of data means.

However, oftentimes, it will seem reasonable that, since domain type data looks the same in almost every table, creating just one such table and reusing it in multiple locations would be a great idea. This is an idea from people who are architecting a relational database who don't really understand relational database architecture (me included, early in my career)—that the more tables there are, the more complex the design will be. So, conversely, condensing multiple tables into a single catchall table should simplify the design, right? That sounds logical, but at one time giving Pauly Shore the lead in a movie sounded like a good idea too.

As an example, consider that I am building a database to store customers and orders. I need domain values for the following:

- Customer credit status
- Customer type
- Invoice status
- Invoice line item back order status
- Invoice line item ship via carrier

Why not just use one generic table to hold these domains, as shown in Figure 8-18.

I agree with you if you are thinking that this seems like a very clean way to implement this from a coding-only standpoint. The problem from a relational coding/implementation standpoint is that it is just not natural to work with in SQL. In many cases, the person who does this does not even think about SQL access. The data in GenericDomain is most likely read into cache in the application and never queried again. Unfortunately, however, this data will need to be used when the data is reported on. For example, say the report writer wants to get the domain values for the Customer table:

```
SELECT *
FROM Customer
  JOIN GenericDomain as CustomerType
    ON Customer.CustomerTypeId = CustomerType.GenericDomainId
      AND CustomerType.RelatedToTable = 'Customer'
      AND CustomerType.RelatedToColumn = 'CustomerTypeId'

  JOIN GenericDomain as CreditStatus
    ON Customer.CreditStatusId = CreditStatus.GenericDomainId
      AND CreditStatus.RelatedToTable = 'Customer'
      AND CreditStatus.RelatedToColumn = ' CreditStatusId'
```


www.it-ebooks.info

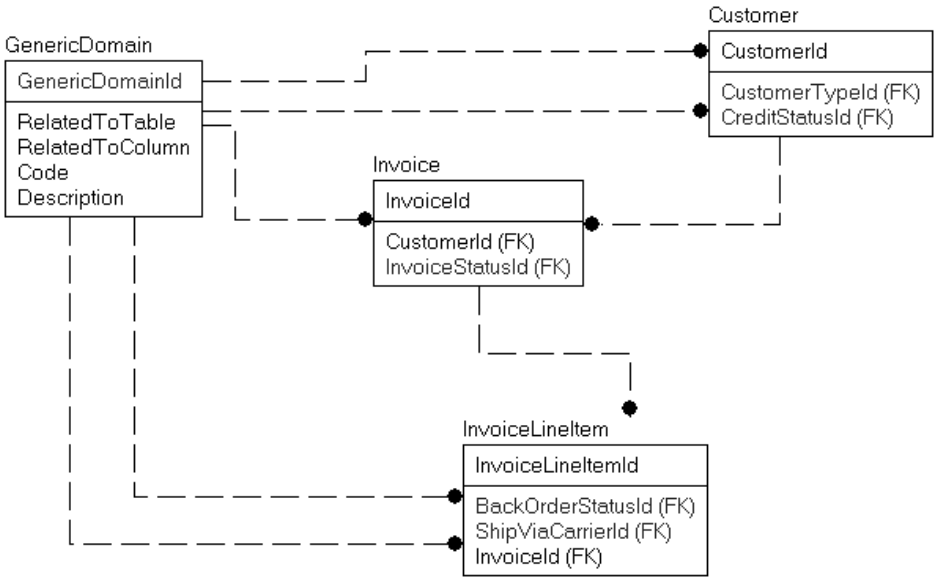


Figure 8-18. *One multiuse domain table*

As you can see, this is far from being a natural operation in SQL. It comes down to the problem of mixing apples with oranges. When you want to make apple pie, you have to strain out only apples so you don't get them mixed. At first glance, domain tables are just an abstract concept of a container that holds text. And from an implementation-centric standpoint, this is quite true, but it is not the correct way to build a database because we never want to mix the rows together as the same thing ever in a query. The litmus test is if you will *never* use a domain except for one table, it should have its own table. You can tell this pretty easily usually, because you will need to specify a table and/or column name to figure out the rows that are applicable.

In a database, the process of normalization as a means of breaking down and isolating data takes every table to the point where one table represents one type of thing and one row represents the existence of one of those things. Every independent domain of values should be thought of as a distinctly different thing from all the other domains (unless it is not, in which case one table will suffice). So, what you do, in essence, is normalize the data over and over on each usage, spreading the work out over time, rather than doing the task once and getting it over with.

Instead of a single table for all domains, you should model it as shown in Figure 8-19.

That looks harder to do, right? Well, it is initially (like for the 5 or 10 minutes it takes to create a few tables). Frankly, it took me longer to flesh out the example tables. The fact is, there are quite a few tremendous gains to be had:

- *Using the data in a query is much Clearer*:

```
SELECT *
FROM   Customer
       JOIN CustomerType
         ON Customer.CustomerTypeId = CustomerType.CustomerTypeId
       JOIN CreditStatus
         ON Customer.CreditStatusId = CreditStatus.CreditStatusId
```


www.it-ebooks.info

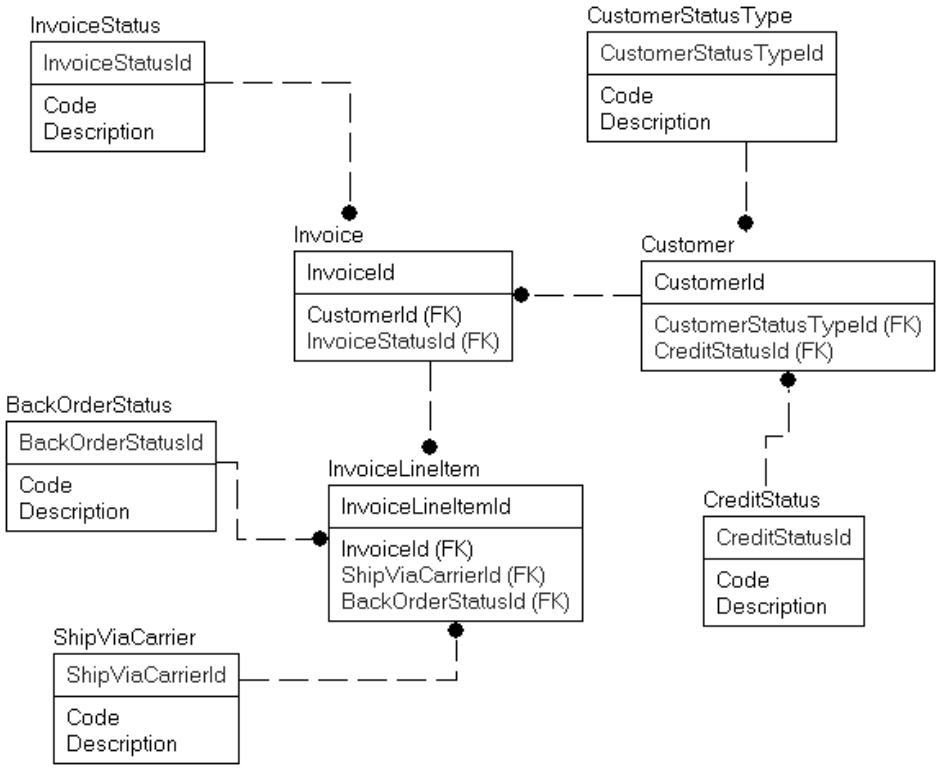


Figure 8-19. *One domain table per purpose*

- *Data can be validated using simple foreign key constraints*: This was something not feasible for the one-table solution. Now, validation needs to be in triggers or just managed solely by the application.
- *Expandability and control*: If it turns out that you need to keep more information in your domain row, it is as simple as adding a column or two. For example, if you have a domain of shipping carriers, you might define a `ShipViaCarrier` in your master domain table. In its basic form, you would get only one column for a value for the user to choose. But if you wanted to have more information—such as a long name for reports, as in "United Parcel Service"; a description; and some form of indication when to use this carrier—you would be forced to implement a table and change all the references to the domain values.
- *Performance considerations*: All of the smaller domain tables will fit on a single page or disk. This ensures a single read (and likely a single page in cache). If the other case, you might have your domain table spread across many pages, unless you cluster on the referring table name, which then could cause it to be more costly to use a nonclustered index if you have many values. In a very large table, it could get to the point where a scan of a larger domain table could get costly where only a very small number of rows is needed.

www.it-ebooks.info

- *You can still make the data look like one table for the application*: There is nothing precluding developers from building a caching mechanism that melds together all the individual tables to populate the cache and use the data however they need it for the application. With some clever use of extended properties, this could be as simple as adding a value to a property and letting a dynamic SQL procedure return all the data. A common concern that developers have is that now they will need 50 editors instead of one. You can still have one editor for all rows, because most domain tables will likely have the same base structure/usage, and if they don't, you will already need to create a new table or do some sort of hokey usage to make the single table design work.

Returning to the basics of design, every table should represent one and only one thing. When you see a column in a table by itself, there should be no question as to what it means, and you certainly shouldn't need to go to the table and figure out what the meaning of a value is.

Some tools that implement an object-oriented view of a design tend to use this frequently, because it's easy to implement tables such as this and use a cached object. One table means one set of methods instead of hundreds of different methods for hundreds of different objects—er, tables. (The fact that it stinks when you go to use it in the database for queries is of little consequence, because generally systems like this don't, at least initially, intend for you to go into the database and do queries, except through special interfaces that take care of this situation for you.)

Generic Key References

In an ideal situation, one table is related to another via a key. However, because the structures in SQL Server don't require constraints or any enforcement, this can lead to interesting relationships occurring. What I am referring to here is the case where you have a table that has a primary key that can actually be a value from several different tables, instead of just one.

For example, consider the case where you have several objects, all of which need a reference to one table. In our sample, say you have a customer relationship management system with `SalesOrders` and `TroubleTickets` (just these two to keep it simple, but in reality, you might have many objects in your database that will fit this scenario). Each of these objects has the need to store journal items, outlining the user's contact with the customer (for example, in the case where you want to make sure not to overcommunicate with a customer!). You might logically draw it up like in Figure 8-20.

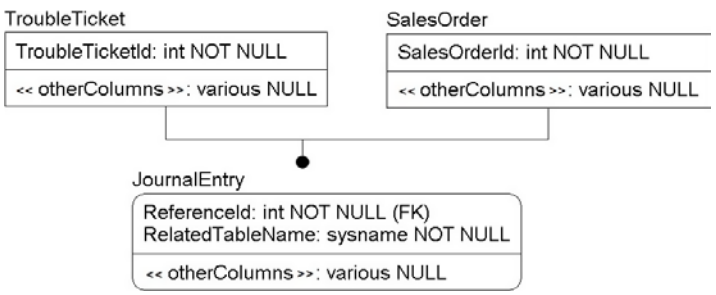


Figure 8-20. *Multiple tables related to the same key*

You might initially consider modeling it like a classic subtype relationship, but it really doesn't fit that mold because you probably can have more than one journal entry per sales order and trouble ticket. Fair enough, each of these relationships is 1–*N*, where *N* is between 0 and infinity (though the customer with infinite journal entries

www.it-ebooks.info

must really hate you). Having all parents relate to the same column is a possible solution to the problem but not a very favorable one. For our table in this scenario, we build something like this:

```
CREATE TABLE SalesOrder
(
        SalesOrderId <int or uniqueidentifier> PRIMARY KEY,
        <other columns>
);
CREATE TABLE TroubleTicket
(
        TroubleTicketId <int or uniqueidentifier> PRIMARY KEY,
        <other columns>
);
CREATE TABLE JournalEntry
(
        JournalEntryId <int or uniqueidentifier>,
        RelatedTableName sysname,
        PRIMARY KEY (JournalEntryId, RelatedTableName)
        <other columns>
);
```

Now, to use this data, you have to indicate the table you want to join to, which is very much a unnatural way to do a join. You can use a universally unique GUID key so that all references to the data in the table are unique, eliminating the need for the specifically specified related table name. However, I find when this method is employed if the `RelatedTableName` is actually used, it is far clearer to the user what is happening.

A major concern with this method is that you cannot use constraints to enforce the relationships; you need either to use triggers or to trust the middle layers to validate data values, which definitely increases the costs of implementation/testing, since you have to verify that it works in all cases, which is something we trust for constraints; even triggers are implemented in one single location.

One reason this method is employed is that it is very easy to add references to the one table. You just put the key value and table name in there, and you are done. Unfortunately, for the people who have to use this for years and years to come, well, it would have just been easier to spend a bit longer and do some more work, because the generic relationship means that using a constraint is not possible to validate keys, leaving open the possibility of orphaned data.

A second way to do this that is marginally better is to just include keys from all tables, like this:

```
CREATE TABLE JournalEntry
(
        JournalEntryId <int or uniqueidentifier> PRIMARY KEY,
        SalesOrderId <int or uniqueidentifier> NULL REFERERENCES
                        SalesOrder(SalesOrderId),
        TroubleTicketId <int or uniqueidentifier> NULL REFERERENCES
                                                TroubleTicket(TroubleTicketId),
        <other columns>
);
```

This is better, in that now joins are clearer and the values are enforced by constraints, but now, you have one more problem (that I conveniently left out of the initial description). What if you need to store some information about the reason for the journal entry? For example, for an order, are you commenting in the journal for a cancelation notice?

There is also the matter of nor concerns, since normalization/usage concerns in that the related values doesn't exactly relate to the key in the same way. It seems like a decent idea that one `JournalEntry` might relate to more than one `SalesOrder` or `JournalEntry`. So, the better idea is to model it more like Figure 8-21.

www.it-ebooks.info

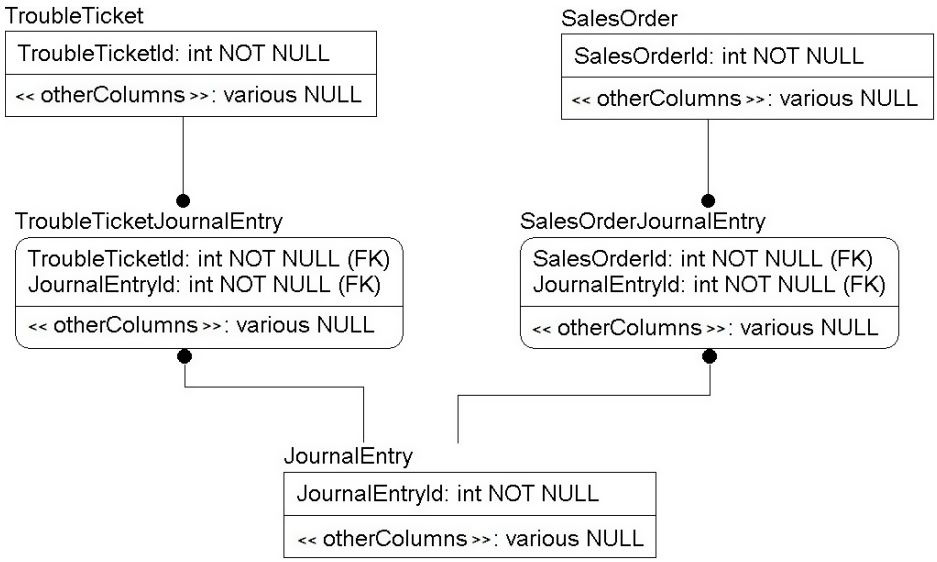


Figure 8-21. *Objects linked for maximum usability/flexibility*

```
CREATE TABLE JournalEntry
(
        JournalEntryId <int or uniqueidentifier> PRIMARY KEY,
        <other columns>
);
CREATE TABLE SalesOrderJournalEntry
(
        JournalEntryId <int or uniqueidentifier>
                        REFERENCES JournalEntry(JournalId),
        SalesOrderId <int or uniqueidentifier>,
                       REFERENCES SalesOrder(SalesOrderId),
        <SalesOrderSpecificColumns>
        PRIMARY KEY (JournalEntryId, SalesOrderId)
);
CREATE TABLE TroubleTicketJournalEntry
(
        JournalEntryId <int or uniqueidentifier>
                        REFERENCES JournalEntry(JournalId),
        TroubleTicketId <int or uniqueidentifier>,
                        REFERENCES TroubleTicket (TroubleTicketId),
        <TroubleTicketSpecificColumns>
        PRIMARY KEY (JournalEntryId, SalesOrderId)
);
```

Note that this database is far more self-documented as well. You can easily find the relationships between the tables and join on them. Yes, there are a few more tables, but that can play to your benefit as well in some scenarios, but most important, you can represent any data you need to represent, in any cardinality or combination of cardinalities needed. This is the goal in almost any design.

www.it-ebooks.info

Overusing Unstructured Data

As much as I would like to deny it, or at least find some way to avoid it, people need to have unstructured notes to store various bits and pieces of information about their data. I will confess that a large number of the tables I have created in my career included some column that allowed users to insert whatever into. In the early days, it was a `varchar(256)` column, then `varchar(8000)` or `text`, and now `varchar(max)`. It is not something that you can get away from, because users need this scratchpad just slightly more than Linus needs his security blanket. And it is not such a terrible practice, to be honest. What is the harm in letting the user have a place to note that the person has special requirements when you go out to lunch?

Nothing much, except that far too often what happens is that notes become a replacement for the types of stuff that I mentioned in the "User-Specified Data" section or, when this is a corporate application, the types of columns that you could go in and create in 10 minutes or about 2 days, including testing and deployment. Once the users do something once and particularly finds it useful, they will do it again. And they tell their buddies, "Hey, I have started using notes to indicate that the order needs processing. Saved me an hour yesterday ." Don't get me wrong, I have nothing against users saving time, but in the end, everyone needs to work together.

See, if storing some value unstructured in a notes column saves the user any time at all (considering that, most likely, it will require a nonindexed search or a one-at-a-time manual search), just think what having a column in the database could do that can be easily manipulated, indexed, searched on, and oblivious to the spelling habits of the average human being. And what happens when a user decides that they can come up with a "better" way and practices change, or, worse, everyone has their own practices?

Probably the most common use of this I have seen that concerns me is contact notes. I have done this myself in the past, where you have a column that contains formatted text something like the following on a `Customer` table. Users can add new notes but usually are not allowed to go back and change the notes.

■ **ContactNotes** 2008-01-11 – Stuart Pidd -Spoke to Fred on the phone. Said that his wangle was broken, referencing Invoice 20001. Told him I would check and call back tomorrow.

2008-02-15 – Stuart Pidd – Fred called back, stating his wangle was still broken, and now it had started to dangle. Will call back tomorrow.

2008-04-12 – Norm Oliser – Stu was fired for not taking care of one of our best customers.

What a terrible waste of data. The proper solution would be to take this data that is being stored into this text column and apply the rigors of normalization to it. Clearly, in this example, you can see three "rows" of data, with at least three "columns." So instead of having a `Customer` table with a `ContactNotes` column, implement the tables like this:

```
CREATE TABLE Customer
(
        CustomerId int CONSTRAINT PKCustomer PRIMARY KEY
        <other columns>
);
CREATE TABLE CustomerContactNotes
(
        CustomerId int,
        NoteTime datetime,
        PRIMARY KEY (CustomerId, NoteTime),
        UserId datatype, --references the User table
        Notes varchar(max)
);
```


www.it-ebooks.info

You might even stretch this to the model we discussed earlier with the journal entries where the notes are a generic part of the system and can refer to the customer, multiple customers, and other objects in the database. This might even link to a reminder system to remind Stu to get back to Fred, and he would not now be jobless. Though one probably should have expected such out of a guy named Stu Pidd (ba boom ching).

Even using XML to store the notes in this structured manner would be an amazing improvement. You could then determine who entered the notes, what the day was, and what the notes were, and you could fashion a UI that allowed the users to add new fields to the XML, right on the fly. What a tremendous benefit to your users and, let's face it, to the people who have to go in and answer questions like this, "How many times have we talked to this client by phone?"

The point of this section is simply this: educate your users. Give them a place to write the random note, but teach them that when they start to use notes to store the same specific sorts of things over and over, their jobs could be easier if you gave them a place to store their values that would be searchable, repeatable, and so on. Plus, never again would you have to write queries to "mine" information from notes.

■ **Tip** SQL Server provides a tool to help search text called Full Text Search. It can be very useful for searching textual data in a manner much like a typical web search. However, it is no replacement for proper design that makes a different column and row from every single data point that the users are typically interested in.

Summary

This chapter was dedicated to expanding the way you think about tables and to giving you some common solutions to problems that are themselves common. I was careful not to get too esoteric with my topics in this chapter. The point was simply to cover some solutions that are a bit beyond the basic table structures I covered in earlier chapters but not so beyond them that the average reader would say "Bah!" to the whole chapter as a waste of time.

The "good" patterns we covered were:

- *Uniqueness*: Simple uniqueness constraints are often not enough to specify uniqueness for "real" data. We discussed going deeper than basic implementation and working through uniqueness scenarios where you exclude values (selective uniqueness), bulk object uniqueness, and discussed the real-world example of trying to piece together uniqueness where you can't be completely sure (like visitors to a web site).
- *Data-driven design*: The goal being to build your databases flexible enough that adding new data to the database that looks and acts like previous values does not require code changes. We do this by attempting to avoid hard-coded data that is apt to change and making columns for typical configurations.
- *Hierarchies*: We discussed several methods of implementing hierarchies, using simple SQL constructs to using `hierarchyId`, as well as an introduction to the different methods that have been created to optimize utilization with a bit of reduction in simplicity.
- *Large binary data*: This pertains particularly to images but could refer any sort of file that you might find in a Windows file system. Storing large binary values allows you to provide your users with a place to extend their data storage.
- *Generalization*: Although this is more a concept than a particular pattern, we discussed why we need to match the design to the users realistic needs by generalizing some objects to the system needs (and not to our nerdy academic desires).

www.it-ebooks.info

We finished up with a section on anti-patterns and poor design practices, including some pretty heinous ones:

- *Undecipherable data*: All data in the database should have some sort of meaning. Users should not have to wonder what a value of 1 means.
- *Overusing unstructured data*: Basically, this hearkens back to normalization, where we desire to store one value per column. Users are given a generic column for notes regarding a given item, and because they have unplanned-for needs for additional data storage, they use the notes instead. The mess that ensues, particularly for the people who need to report onthis data, is generally the fault of the architect at design time to not give the users a place to enter whatever they need, or to be fair, the users changing their needs over time and adapting to the situation rather than consulting the IT team to adjust the system to their ever changing needs.
- *One domain table to cover all domains*: This is yet another normalization issue, because the overarching goal of a database is to match one table with one need. Domain values may seem like one thing, but the goal should be that every row in a table is usable in any table it is relatable to.
- *Generic key references*: It is a very common need to have multiple tables relate to another. It can also be true that only one table should be related at a time. However, every column should contain one and only one type of data. Otherwise, users have no idea what a value is unless they go hunting.

Of course, these lists are not exhaustive of all of the possible patterns out there that you should use or not use, respectively. The goal of this chapter was to help you see some of the common usages of objects so you can begin to put together models that follow a common pattern where it makes sense. Feedback, particularly ideas for new sections, is always desired at louis@drsql.org.

www.it-ebooks.info

CHAPTER 9

Database Security and Security Patterns

"If you want total security, go to prison. There you're fed, clothed, given medical care and so on. The only thing lacking . . . is freedom."

—Dwight D. Eisenhower

There are so many threats to your security that it is essential to remain ever vigilant—without ending up with your server in a bunker of lead wearing a tinfoil hat protecting data by keeping it completely inaccessible to any human eyes. Business needs connectivity to customers, and customers need connectivity to their data. Security is one of the most important tasks when setting up and creating a new application, yet it is often overlooked and dealt with late in the application building process. Whether or not this is acceptable is generally up to your requirements and how your application will be built, but at one point or another, your application team must take the time to get serious about security. Over and over, stories in the news report data being stolen, and the theft is inevitably due to poor security. In the last edition of this book, I used the example of an election official's stolen laptop in my home city of Nashville, Tennessee; names, addresses, and partial social security numbers were stolen. Since then, there has been a steady stream of such stories, and probably the most high profile has been Sony's Playstation network getting hacked and being down for months. Hence, if you are the architect of a database system that holds personal and private information, it could be you who becomes jobless with a stain on your career the size of the Mojave desert and possibly quite unemployed if it turns out to be your fault that data leaked out into the hands of some junkie looking for stuff to hock.

Security is on the minds of almost every company today, as evidenced by the constant stream of privacy policies that we continue to come across these days. They're *everywhere* and if your company does business with anyone, it likely has one too. Let's be clear: for far too many organizations, most security is implemented by hoping average users are as observant as Dorothy and her shoes. They have a lot of power if they just were adventurous enough to open a tool like Management Studio and start clicking around (or perhaps clicking their ruby slipper heels together three times.) Of course, fear is a reasonably good motivator for sticking to the marked path, and most average users aren't too adventurous in the first place (work the help desk for a week, and you will know exactly what I mean). If they were, they'd not only discover how to fix the same problem they had yesterday, but also may just find that they have incredible power to see more than they need to see or to get back home to Kansas in the blink of an eye.

www.it-ebooks.info

In this chapter, we will be covering the following topics:

- *Database security prerequisites*: We will cover some of the fundamentals that you need to understand before dealing with database-level access.
- *Database securables*: Once you are in the context of a database, you have a lot of built-in control over what users can access. In this section, we will cover what they are.
- *Controlling access to data via T-SQL coded objects*: We will look beyond direct access to data, at how you can restrict access to data in more granular ways using T-SQL procedures, views, and so on.
- *Crossing database lines*: Databases are ideally independent containers, but on occasion, you will need to access data that is not stored within the confines of the database. In this section, we will cover some of the caveats when implementing cross database access.
- *Obfuscating data*: Often, you cannot prevent a user from having access to some data, but you want the program to be able to decode the data only situationally. This is particularly important for personally identifiable data or financial data, so we encrypt the data to keep eyes out except where allowable.
- *Monitoring and auditing*: Turning on a "security camera" and watch what people are doing is sometimes the only real way to verify that you can provide adequate security, and in many cases you will do this *and* the aforementioned items.

Overall, we will cover a solid sampling of what you will need to secure your data but not the complete security picture, especially if you start to use some of the features of SQL Server that we are not covering in this book (Service Broker to name one). The goal of this chapter will be to shine a light on what is available, demonstrate some of the implementation patterns you may use, and then let you dig in for your exact needs.

I should also note that not everyone will use many, if any, of the guidelines in this chapter in their security implementations. Often, the application layer is left to implement the security alone, showing or hiding functionality from the user. This approach is common, but it can leave gaps in security, especially when you have to give users ad hoc access to the data or you have multiple user interfaces that have to implement different methods of security. My advice is to make use of the permissions in the database server as much as possible. However, having the application layer control security isn't a tremendous hole in the security of the organization, as long as the passwords used are seriously complex, encrypted, and extremely well guarded and ideally the data is accessed using Windows Authentication from the middle tier.

Database Access Prerequisites

In this initial section of this chapter, we are going to cover a few prerequisites that we will need for the rest of this chapter on database security. As a programmer, I have generally only been an advisor on how to configure most of the server beyond the confines of the individual database. Setting up the layers of security at the SQL Server instance and Windows Server level is not tremendously difficult, but it is certainly outside of the scope of this book on database design.

As a bit of an introduction to the prerequisites, I am going to cover a few topics to get you started on your way to implementing a secure environment:

- *Guidelines for server security*: In this section, I will cover some of the things you can use to make sure your server is configured to protect against outside harm.
- *Principals and securables*: All security in SQL Server is centered around principals (loosely, logins and users) and securables (stuff that you can limit access to).

www.it-ebooks.info

- *Connecting to the server*: With changes in SQL Server 2012, there are now multiple ways to access the server. We will cover these in this section.
- *Impersonation*: Using the `EXECUTE AS` statement, you can "pretend" you are a different security principal to use the other users' security. It is a very important concept for testing security that we will use often in this chapter.

Guidelines for Server Security

Even as strictly a database architect/programmer, you may need to set up, or at least validate, the security of your SQL Server installation. The following bulleted list contains some high-level characteristics you will want to use to validate the security of the server to protect your system from malicious hackers. It is not an exhaustive list, but it is a good start nonetheless:

- Strong passwords are applied to all accounts, both Windows Authentication and SQL Server authentication style (server and contained database style, new to SQL Server 2012)—certainly, all universally known system accounts have very strong passwords (such as `sa`, if you haven't changed the name). Certainly, there are no blank passwords for any accounts!
- SQL Server isn't sitting unguarded on the Web, with no firewall and no logging of failed login attempts.
- The `guest` user has been removed from all databases where it isn't necessary.
- Care has been taken to guard against SQL injection by avoiding query strings whereby a user could simply inject `SELECT name FROM sys.sql_logins` and get a list of all your logins in a text box in your application that should display something like toothpaste brands. (Chapter 13 mentions SQL injection again; there I contrast ad hoc SQL with stored procedures.)
- Application passwords are secured/encrypted and put where they can be seen by only necessary people (such as the DBA and the application programmers who use them in their code). The password is encrypted into application code modules when using application logins.
- You've made certain that few people have file-level access to the server where the data is stored and, probably more important, where the backups are stored. If one malicious user has access to your backup file (or tape if you are still living in the past), that person has access to your data by simply attaching that file to a different server, and you can't stop him or her from accessing the data (even encryption isn't 100 percent secure if the hacker has virtually unlimited time).
- You have taken all necessary precautions to make sure that the physical computer where the data is stored cannot be taken away as a whole. Even things like encryption aren't completely effective if the data needed to decrypt the values is available on one of the machines that has been stolen along with the machine with the encrypted values.
- Your SQL Server installation is located in a very secure location. A Windows server, just like your laptop, is only as secure as the physical box. Just like on any spy TV show, if the bad guys can access your physical hardware, they could boot to a CD or USB device and have access to your hard disks (note that using transparent data encryption (TDE) can help in this case).

www.it-ebooks.info

- All features that you are not using are turned off. To make SQL Server as secure as possible out of the box, many features are disabled by default and have to be enabled explicitly before you can use them. For example, remote administrator connections, Database Mail, CLR programming, and others are all off by default. You can enable these features and others using the `sp_configure` stored procedure.

Principals and Securables

At the very core of security in SQL Server are the concepts of principals and securables. *Principals* are those objects that may be granted permission to access particular database objects, while *securables* are those objects to which access can be controlled. Principals can represent a *specific user*, a role that may be adopted by multiple users, or an application, certificate, and more. There are three sorts of SQL Server principals that you will deal with:

- *Windows principals*: These represent Windows user accounts or groups, authenticated using Windows security.
- *SQL Server principals*: These are server-level logins or groups that are authenticated using SQL Server security.
- *Database principals*: These include database users, groups, and roles, as well as application roles.

Securables are the database objects to which you can control access and to which you can grant principals permissions. SQL Server distinguishes between three scopes at which different objects can be secured:

- *Server scope*: Server-scoped securables include logins, HTTP endpoints, event notifications, and databases. These are objects that exist at the server level, outside of any individual database, and to which access is controlled on a server-wide basis.
- *Database scope*: Securables with database scope are objects such as schemas, users, roles, and CLR assemblies, DDL triggers, and so on, which exist inside a particular database but not within a schema.
- *Schema scope*: This group includes those objects that reside within a schema in a database, such as tables, views, and stored procedures. A SQL Server 2005 and later schema corresponds roughly to the owner of a set of objects (such as `dbo`) in SQL Server 2000.

These concepts will come into play in all of the following sections as we walk through the different ways that you will need to secure the data in the database. You can then grant or deny usage of these objects to the roles that have been created. SQL Server uses three different security statements to give or take away rights from each of your roles:

- `GRANT`: Gives the privilege to use an object.
- `DENY`: Denies access to an object, regardless of whether the user has been granted the privilege from any other role.
- `REVOKE`: Used to remove any `GRANT` or `DENY` permissions statements that have been applied to an object. This behaves like a delete of an applied permission, one either granted or denied.

Typically, you'll simply give permissions to a role to perform tasks that are specific to the role. `DENY` is then used only in "extreme" cases, because no matter how many other times the user has been granted privileges to an object, the user won't have access to it while there's one `DENY`.

www.it-ebooks.info

For a database or server right, you will use syntax like

```
GRANT <privilege> TO <principal> [WITH GRANT OPTION];
```

The `WITH GRANT OPTION` will allow the principal to grant the privilege to another principal.

For the most part, this book will deal primarily with database object privileges, as database and server privileges are almost always an administrative consideration. They allow you to let principals create objects, drop objects, do backups, view metadata, and so on.

For database objects, there is a minor difference in the syntax, in that the securable that you will be granting rights to will be specified. For example, to grant a privilege on a securable in a database, the command would be as follows:

```
GRANT <privilege> ON <securable> to <principal> [WITH GRANT OPTION];
```

Next, if you want to remove the privilege, you have two choices. You can either `REVOKE` the permission, which just deletes the granted permission, or you can `DENY` the permission. Execute the following:

```
REVOKE <privilege> FROM <securable> to <principal>;
```

I haven't covered role membership yet (it's covered later in this chapter), but if the user were a member of a role that had access to this object, the user would still have access. However, execute the following code:

```
DENY <privilege> ON <securable> to <principal>;
```

The use of `DENY` will prohibit the principal from using the securable, even if they have also been granted access by means of another securable. To remove `DENY`, you again use the `REVOKE` command. This will all become clearer when I cover roles later in this chapter, but in my experience, `DENY` isn't a typical thing to use on a principal's privilege set. It's punitive in nature and is confusing to the average user. More commonly, users are given rights and not denied access.

Another bit of notation you will see quite often is to denote the type of securable before the securable where it is not the default. For objects that show up in `sys.objects` that have security granted to them (from Books Online, these are table, view, table-valued function, stored procedure, extended stored procedure, scalar function, aggregate function, service queue, or synonym), you can simply reference the name of the object:

```
GRANT <privilege> ON <securable> to <database principal>;
```

For other types of objects, such as schemas, assemblies, and search property lists, to name a few, you will specify the type in the name. For example, for a schema `GRANT`, the syntax is

```
GRANT <privilege> ON SCHEMA::<schema securable> to <database principal>;
```

Note that, for objects, you can also use the a prefix of `OBJECT::`, as in the following:

```
GRANT <privilege> ON OBJECT::<securable> to <database principal>;
```

Connecting to the Server

Before we finally get the database security, we need to cover accessing the server. Prior to SQL Server 2012, there was a single way to access a database. This method is still pretty much the norm and is basically as follows: A login principal is defined that allows a principal to access the server using Windows credentials, a login that is managed in the SQL Server instance (known as SQL Server authentication), or one of several other methods including a certificate or an asymmetric key. The login is then mapped to a user within the database to gain access.

www.it-ebooks.info

The additional method in SQL Server 2012 uses a new concept of a contained database (CDB). I will cover the broader picture and a bit of the management of CDB as a whole later in the chapter when I cover cross-database security, but I do need to introduce the syntax and creation of the database here as it is, from a coding standpoint, largely a security question. Contained databases in SQL Server 2012 are the initial start of making databases essentially standalone containers that can be moved from server to server with little effort (and likely eventually to Azure as well).

In this section, I will provide two examples of connecting to the server:

- Using the classic approach of a login and user
- Access the database directly using the containment model

Using Login and User

To access the server, we will create a server principal known as a login. There are two typical methods that you will use to create almost all logins. The first method is to map a login to a Windows Authentication principal. This is done using the `CREATE LOGIN` statement. The following example would create the login I have on my laptop for writing content:

```
CREATE LOGIN [DENALI-PC\AlienDrsql] FROM WINDOWS
  WITH DEFAULT_DATABASE=tempdb, DEFAULT_LANGUAGE=us_english;
```

The name of the login is the same as the name of the Windows principal, which is how they map together. So on my local virtual machine named `DENALI-PC`, I have a user named `AlienDrsql` (I have an Alienware PC, hence the name; I am not a weirdo; I promise.) The Windows principal can be a single user or a Windows group. For a group, all users in the group will gain access to the server in the same way and have the exact same permission set. This is, generally speaking, the most convenient method of creating and giving users rights to SQL Server.

The second way is to create a login with a password:

```
CREATE LOGIN [Fred] WITH PASSWORD=N'password' MUST_CHANGE, DEFAULT_DATABASE=[tempdb],
  DEFAULT_LANGUAGE=[us_english], CHECK_EXPIRATION=ON, CHECK_POLICY=ON;
```

If you set the `CHECK_POLICY` setting to `ON`, the password will need to follow the password complexity rules of the Windows server it is created on, and `CHECK_EXPIRATION`, when set to `ON`, will require the password to be changed based on the policy of the Windows server as well. Generally speaking, the most desirable method is to use Windows Authentication for the default access to the server where possible, since keeping the number of passwords a person has to a minimum makes it less likely for them to tape the password up on the wall for all to see. Of course, using Windows Authentication can be troublesome in some cases where SQL Server are located in a DMZ with no trust between domains so you have to resort to SQL Server authentication, so use very complex passwords and (ideally) change them often.)

In both cases, I defaulted the database to `tempdb`, because it requires a conscious effort to go to a user database and start building, or even dropping, objects. However, any work done in `tempdb` is deleted when the server is stopped. This is actually one of those things that may save you more times than you might imagine. Often, a script gets executed and the database is not specified, and a bunch of data gets created—usually in `master` (the default database if you haven't set one explicitly…so the default default database.) I have built more test objects on my local SQL Server in `master` over the years than I can count.

Once you have created the login, you will need to do something with it. If you want to make it a system administrator–level user, you could add it to the `sysadmin` group, which is something that you will want to do on your local machine with your default user (though you probably already did this when you were installing the

www.it-ebooks.info

server and working though the previous chapters, probably during the installation process, possibly without even realizing that was what you were doing):

```
ALTER SERVER ROLE [sysadmin] ADD MEMBER [DENALI-PC\AlienDrsql];
```

■ **Tip** Members of the `sysadmin` role basically bypass all rights checks on the server and are allowed to do anything. It is important to make sure you always have one `sysadmin` user that someone has the credentials for. It may sound obvious, but many a server has been reinstalled after all `sysadmin` users have been dropped or lost their passwords.

You can give users rights to do certain actions using server permissions. For example, you might give the `FRED` user rights to `VIEW SERVER STATE` (which lets you run Dynamic Management Views, for example) using:

```
GRANT VIEW SERVER STATE to [Fred];
```

And new in SQL Server 2012, you can create user-defined server roles. For example, if you want to give support people rights to `VIEW SERVER STATE` and `VIEW ANY DATABASE` (letting them see the structure of all databases) rights, you could create a server role:

```
CREATE SERVER ROLE SupportViewServer;
```

Grant the role the rights desired:

```
GRANT VIEW SERVER STATE to SupportViewServer;
GRANT VIEW ANY DATABASE to SupportViewServer;
```

And add the login to the server role:

```
ALTER SERVER ROLE SupportViewServer ADD MEMBER Fred;
```

Once we have our login created, the next step is to set up access to a database (unless you used `sysadmin`, in which case you have unfettered access to everything on the server). For example, let's create a simple database. For the remainder of this chapter, I will expect that you are a using a user who is a member of the `sysadmin` server role as the primary user, much as we have for the entire book, except when we are testing some code, and it will be set up in the text as to what we are doing. So we create database `ClassicSecurityExample`:

```
CREATE DATABASE ClassicSecurityExample;
```

Next, we will create another login, using SQL Server Authentication. Most logins we will create in the book will be SQL Server Authentication to make it easier to test the user. We will also keep the password simple (`CHECK_POLICY`) and not require it to be changed (`CHECK_EXPIRATION`) to make our examples easier:

```
CREATE LOGIN Barney WITH PASSWORD=N'password', DEFAULT_DATABASE=[tempdb],
  DEFAULT_LANGUAGE=[us_english], CHECK_EXPIRATION=OFF, CHECK_POLICY=OFF;
```

Login using the user in Management Studio into a query window:

www.it-ebooks.info

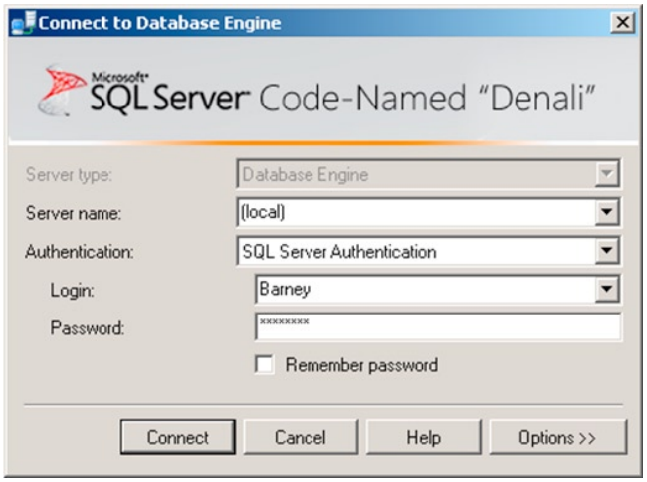


Figure 9-1. *Logging in using test user*

Next, try to execute USE statement to change context to the `ClassicSecurityExample` database:

```
USE ClassicSecurityExample;
```

You will receive the following error:

```
Msg 916, Level 14, State 1, Line 1
The server principal "Barney" is not able to access the database "ClassicSecurityExample"
under the current security context.
```

Your database context will remain in `tempdb`, since this is the default database we set up for the user. Going back to the window where you are in the `sysadmin` user context, we need to enable the user to access the database. There are two ways to do this, the first being to give the `guest` user rights to connect to the database:

```
USE ClassicSecurityExample;
GO
GRANT CONNECT to guest;
```

If you go back to the connection where the user Barney is logged in, you will find that Barney can now access the `ClassicSecurityExample` database—as can any other login in your system. You can apply this strategy if you have a database that you want all users to have access to, but it is generally not the best idea under most circumstances.

So, let's remove this right from the `guest` user using the `REVOKE` statement:

```
REVOKE CONNECT TO guest;
```

Going back to the window where you have connected to the database as Barney, you will find that executing a statement is still allowed, but if you disconnect and reconnect, you will not be able to access the database. Finally, to give server principal `Barney` access to the database, we will create a user in the database and grant it the right to connect:

```
USE ClassicSecurityExample;
GO
```


www.it-ebooks.info

```
CREATE USER BarneyUser FROM LOGIN Barney;
GO
GRANT CONNECT to BarneyUser;
```

Going back to the query window in the context of Barney, you will see that you can connect to the database, and using a few system functions, you can see your server and database security contexts in each.

```
USE ClassicSecurityExample;
GO
SELECT SUSER_SNAME() as server_principal_name, USER_NAME() as database_principal_name;
```

This will return:

```
server_principal_name     database_principal_name
--------------------      -----------------------
Barney                    BarneyUser
```

Executing this in your system administrator connection, you will see:

```
server_principal_name   database_principal_name
--------------------    -----------------------
DENALI-PC\AlienDrsql    dbo
```

The server principal will be the login you used, and the database principal will always be dbo (the database owner), as the system administrator user will always be mapped to the database owner. Now, this is the limit of what we are covering in this section, as you are now able to connect to the database. We will cover what you can do in the database after we cover connecting to the database with a contained database.

Using the Contained Database Model

A tremendous paradigm shift has occurred since I first started writing about database design and programming, and this is virtualization. Even back with SQL Server 2008, the advice would have been strongly against using any sort of virtualization technology with SQL Server, and now, even at the nonprofit I work for, we have nearly everything running on virtualized Windows hardware. One of the many tremendous benefits of virtualization is that you can move around your virtual computer and/or servers within your enterprise to allow optimum use of hardware.

And since SQL 2008 was presented, another paradigm shift has begun with what is referred to as the *cloud*, where instead of housing your own hardware and servers, you put your database on a massive server such that, on average, their hardware is used quite constantly but the users don't exactly feel it. I haven't made too big of a deal about the cloud version of SQL Server (SQL Server Azure) in this book, largely because it is just (simplifying quite a bit, naturally), a relational database that you use over the WAN instead of the LAN. I expect that, although I will just touch a bit on the subject for this book, in the next edition of this book, the cloud will be far more prevalent.

To that end, Microsoft has added the beginnings of what is going to be a very important step in making databases easy to move from local to cloud with ease called contained databases. Where applicable, I will note some of the places where contained database security is different that the classic model, which is mostly concerning accessing external objects.

Our first step is to create a new database that we will set CONTAINMENT = PARTIAL. For SQL Server 2012, there are two models: OFF, which I am referring to as classic model, and PARTIAL, which will give you a few benefits

www.it-ebooks.info

(like temporary object collation defaulting to the partially contained databases rather than the server). Later versions of SQL Server will likely include a fully contained model that will be almost completely isolated from other databases in most ways.

The way you will connect to the database is a fundamental change, and just like filestream we discussed in previous chapters, this means a security point that is going to be turned off by default. Hence, the first thing we will do is configure the server to allow new connections using what is called contained database authentication using `sp_configure`:

```
EXECUTE sp_configure 'contained database authentication', 1;
GO
RECONFIGURE WITH OVERRIDE;
```

You should get a message telling you that the value was changed, either from 0 to 1 or 1 to 1, depending on if the server is already set up for the contained authentication. Next, create the database. You do not set the containment properties in the `CREATE DATABASE` statement, so you will create a database just like any other database:

```
CREATE DATABASE ContainedDBSecurityExample;
```

The next step is to set the containment model using an `ALTER DATABASE` statement, which you will do in the context of the database:

```
USE ContainedDBSecurityExample;
GO
-- set the contained database to be partial
ALTER DATABASE ContainedDBSecurityExample SET CONTAINMENT = PARTIAL;
```

Next, we will create a user, in this case referred to as a contained user. Contained users are basically a hybrid of login and user, and they are created using the `CREATE USER` statement, which is a bit regrettable, as the syntaxes are different (you will be warned if you try to use the wrong syntax). Books Online lists 11 variations of the `CREATE USER` syntax, so you should check it out if you need a different sort of user!

The first case we will use is a new SQL Server authentication user that logs into the database directly with a password that exists in the system catalog tables in the database. You must be in the context of the database (which we set earlier), or you will get an error telling you that you can only create a user with a password in a contained database.

```
CREATE USER WilmaContainedUser WITH PASSWORD = 'p@ssword1';
```

You can also create a Windows Authentication user in the following manner as long as a corresponding login does not exist. So the following syntax is correct, but on my computer, this fails because that user already has a login defined:

```
CREATE USER [DENALI-PC\AlienDrsql];
```

Since that user already has a login, you will get the following error:

```
Msg 15063, Level 16, State 1, Line 1
The login already has an account under a different user name.
```

presumably because it has the same security context, and it would default to using the server rights, with the default database set (as I will demonstrate in the next paragraph!). But again, for demonstration purposes, we will be using SQL Server Authentication to make the process easier.

Next, we will connect to the database in SSMS using the contained user we previously created named `WilmaContainedUser` with password `p@ssword1`. To do this, you will specify the server name, choose SQL Server Authentication, and set the username and password:

www.it-ebooks.info

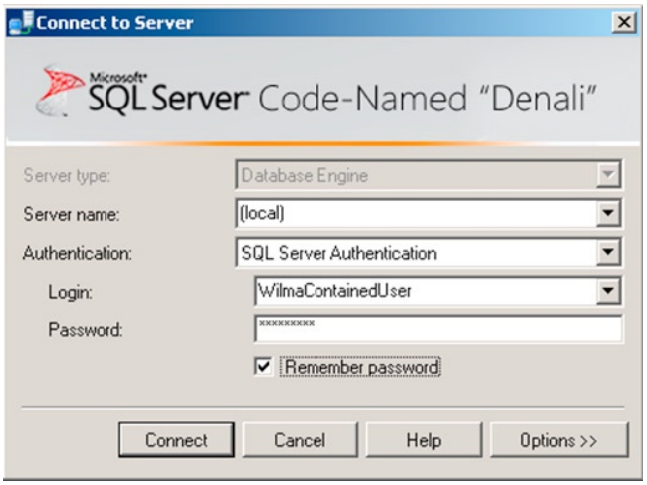


Figure 9-2. *Demonstrating logging into a contained user*

Next, click the Options button. Go to the connection properties tab, and enter the name of the contained database as seen in Figure 9-3.

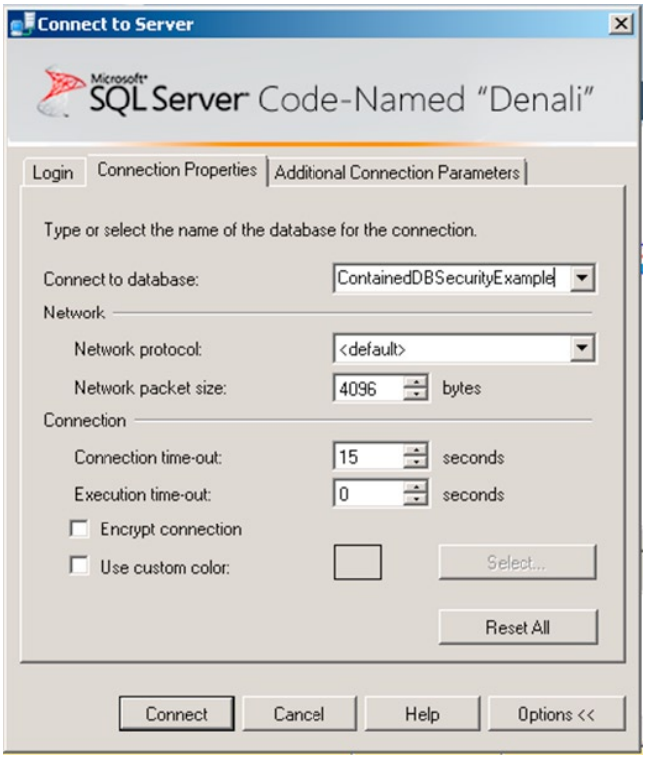


Figure 9-3. *Enter the name of the database in the blank*

www.it-ebooks.info

You will need to know the name since the security criteria you are using will not have rights to the metadata of the server, so if you try to browse the database with the login you have supplied, it will give you the error you can see in Figure 9-4.

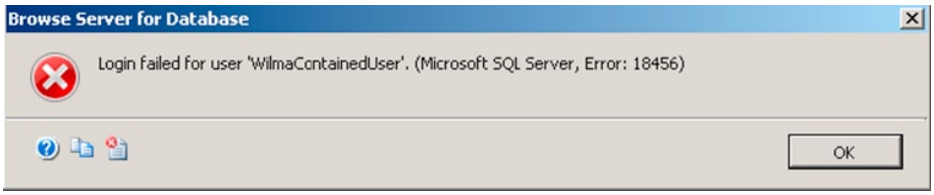


Figure 9-4. *Error trying to browse for name of contained database*

Now, as you will see in Object Explorer, the server seems like it is made up of a single database, as shown in Figure 9-5.

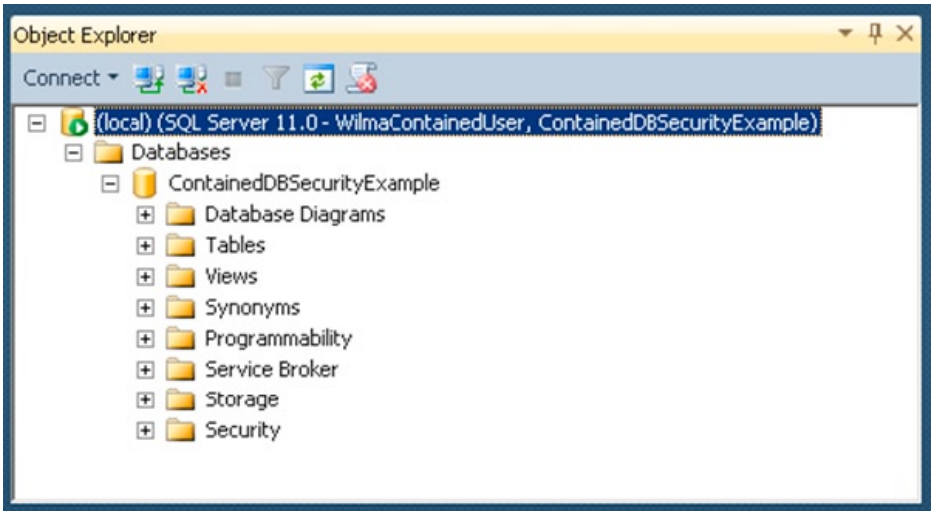


Figure 9-5. *Contained database Object Explorer in SSMS*

After this point in the process, you will be in the context of a database, and everything will be pretty much the same whether the database is partially contained or completely uncontained. The big difference is that in the drop-down list of databases you will have the current database (`ContainedDBSecurityExample`), and `master` and `tempdb`. At this point, you are in the context of the database just like in the previous section on the classic security model.

You cannot create a contained user in an uncontained database, but you can still create a user linked to a login in a contained database. For example, you could create a new login:

```
CREATE LOGIN Pebbles WITH PASSWORD = 'BamBam01$';
```

Then link that user to the login you have created:

```
CREATE USER PebblesUnContainedUser FROM LOGIN Pebbles;
```


www.it-ebooks.info

Obviously, this begins to defeat the overarching value of a contained database, which is to make the database portable without the need to reconcile logins on one server to a login on another, but rather to be immediately usable with the same users (with the caveat that the Windows Authentication user will have to be able to connect to the authenticating server).

Note that you can switch a contained database back to not being contained but you cannot have any contained database principals in it. If you try to set the `ContainedDbSecurityExample` database back to uncontained:

```
ALTER DATABASE ContainedDbSecurityExample SET CONTAINMENT = none;
```

You will get another excellent error message that (unless you have seen it before) will undoubtedly cause you to scratch your head:

```
Msg 33233, Level 16, State 1, Line 1
You can only create a user with a password in a contained database.
Msg 5069, Level 16, State 1, Line 1
ALTER DATABASE statement failed.
```

If you need to make this database uncontained, you will need to drop the contained users, which you can identify with the following list:

```
SELECT  name
FROM    sys.database_principals
WHERE   authentication_type_desc = 'DATABASE';
```

In our database, this returns

```
Name
------------------
WilmaContainedUser
```

Drop this user, and you would then be able to turn containment off for this database. Later in this chapter, we will come back to the topic of containment when we cover cross database access (and in the case of containment, working to protect against it to keep databases more portable).

Impersonation

The ability to pretend to be another user or login is fairly important when it comes to testing security. Impersonation is, in fact, one of the most important tools you will need when you are testing your security configuration. After some code has been migrated to production, it is common to get a call from clients who claims that they cannot do something that you think they really ought to be able to do. Since all system problems are inevitably blamed on the database first, it is a useful trick to impersonate the user and then try the questioned code in Management Studio to see whether it is a security problem. If the code works in Management Studio, your job is almost certainly done from a database standpoint, and you can point your index finger at some other part of the system. You can do all of this without knowing their passwords, as you are either the `sysadmin` user or have been granted rights to impersonate the user.

To demonstrate security in a reasonable manner on a single SQL Server connection, I will use a feature that was new to SQL Server 2005. In 2000 and earlier, if the person with administrator rights wanted to impersonate another user, he or she used `SETUSER` (I still see people use `SETUSER`, so I feel I still need to mention it here). Using `SETUSER`, you can impersonate any server or a database principal, and you get all rights that user has (and consequently lose the rights you previously had). You can go back to the previous

www.it-ebooks.info

security context by executing REVERT. The only downside is that when you try to impersonate a Windows Authentication–based principal, you cannot do this disconnected from the domain where the principal was created.

■ **Note** For non-dbo or sa users to use EXECUTE AS, they must have been granted IMPERSONATE permissions on the specified login name or username that they are trying to impersonate. You can even impersonate a sysadmin-level user if you have been granted such rights.

As an example, I'll show a way that you can have a user impersonating a member of the server-system sysadmin role. Using impersonation in such a way takes some getting used to, but it certainly makes it easier to have full sysadmin power only when it's needed. As said previously, there are lots of server privileges, so you can mete out rights that are needed on a day-to-day basis and reserve the "dangerous" ones like DROP DATABASE only for logins that you have to impersonate.

As an example (and this is the kind of example that I'll have throughout this chapter), we first create a login that we never expect to be logged into directly. I use a standard login, but you could map it to a certificate, a key, a Windows user, or whatever. Standard logins make it much easier to test situations and learn from them because they're self-contained. Then, we add the login to the sysadmin role. You probably also want to use a name that isn't so obviously associated with system administration. If a hacker got into your list of users somehow, the name 'itchy' wouldn't so obviously be able to do serious damage to your database server, as would a name like 'Merlin'.

```
USE master;
GO
CREATE LOGIN system_admin WITH PASSWORD = 'tooHardToEnterAndNoOneKnowsIt',CHECK_POLICY=OFF;
EXEC sp_addsrvrolemember 'system_admin','sysadmin';
```

Then, we create a regular login and give rights to impersonate the system_admin user:

```
CREATE LOGIN louis with PASSWORD = 'reasonable', DEFAULT_DATABASE=tempdb,CHECK_POLICY=OFF;

--Must execute in master Database
GRANT IMPERSONATE ON LOGIN::system_admin TO louis;
```

■ **Caution** You might not want to execute this code on your instance unless you are doing this isolated from production code. The passwords I used (and will use) are far simpler than your production ones will be. For example, the one that was tooHardToEnterAndNoOneKnowsIt would actually be something more like a random string of letters, numbers, and special characters. Some of my current sa passwords have been over 50 characters long and filled with special characters that can only feasibly be pasted to be used.

We log in as louis and try to run the following code (in Management Studio, you can just right-click in the query window to use the Connection/Change Connection context menu and use a standard login):

```
USE ClassicSecurityExample;
```


www.it-ebooks.info

The following error is raised:

```
Msg 916, Level 14, State 1, Line 1
The server principal "louis" is not able to access the database "ClassicSecurityExample" under
the current security context.
```

Now, we change security context to the `system_admin` user (note that you cannot use `EXECUTE AS LOGIN` when you are in the context of a contained database user):

```
EXECUTE AS LOGIN = 'system_admin';
```

We now have control of the server in that window as the `system_admin` user! To look at the security context, you can use several variables/functions:

```
USE     ClassicSecurityExample;
GO
SELECT user as [user], system_user as [system_user],
       original_login() as [original_login];
```

This returns the following result:

```
user    system_user     original_login
----    ------------    --------------
dbo     system_admin    louis
```

The columns mean

- `user`: The database principal name of context for the user in the database
- `system_user`: The server principal name of context for the login
- `original_login()`: The login name of the server principal who actually logged in to start the connection (This is an important function that you should use when logging which login performed an action.)

Then, you execute the following code:

```
REVERT --go back to previous security context
```

We see the following result:

```
Msg 15199, Level 16, State 1, Line 1
The current security context cannot be reverted. Please switch to the original database where
'Execute As' was called and try it again.
```

You started in `tempdb`, so you use the following code:

```
USE tempdb;

REVERT;
SELECT user as [user], SYSTEM_USER as [system_user],
       ORIGINAL_LOGIN() as [original_login];
```


www.it-ebooks.info

This now returns the following result:

```
user     system_user     original_login
-----    -----------     --------------
guest    louis           louis
```

Impersonation gives you a lot of control over what a user can do and allows you to situationally play one role or another, such as creating a new database. I'll use impersonation to change security context to demonstrate security concepts.

■ **Note** The user here is `guest`, which is a user I recommend that you consider disabling in every nonsystem database unless it is specifically needed. Disable `guest` by executing `REVOKE CONNECT FROM GUEST`. You cannot disable the guest user in the `tempdb` or `master` database, because users must have access to these databases to do any work. Trying to disable `guest` in these databases will result in the following message: `"Cannot disable access to the guest user in master or tempdb"`.

Using impersonation, you can execute your code as a member of the `sysadmin` server or `db_owner` database role and then test your code as a typical user without opening multiple connections (and this technique makes the sample code considerably easier to follow). Note that I have only demonstrated impersonating a login, but you can also impersonate users, which we will use along with impersonating a login throughout the rest of this chapter. Note that there are limitations on what you can do when using impersonation. For a full treatment of the subject, check in Books Online under the "`EXECUTE AS`" topic.

Database Securables

Now that we have covered access to the server and/or database, your users are going to need the ability to do something in the database they now have access to. In the previous section, we covered getting into the context of a database, and in this section, we are going to cover the different ways you can now use the database principals you have created.

Permissions to use data securables are rights granted (or denied) to a principal to access some securable. I'll cover the basics of database permissions for a foundation of best practices. Taken to the extreme, an extensively large set of things are considered securable, especially at the server level, but over 90 percent of security for the average database programmer/architect (and certainly a data architect, as is the main focus of this book) is securing tables, views, functions and procedures, and this is what's primarily interesting from a database-design standpoint.

At the database level, there are two main types of principals: the *user* and the *role*. We covered the user in the previous section and whether you use Windows Authentication or standard authentication, or if you use the classic or the containment model, the database implementation will be essentially the same.

The next principal is the role, which is a way to set up different functional roles and then assign a user or another role to it. The very best practice for assigning security to database principals is to nearly always use roles, even if you have only a single user in a role. This practice may sound like more work, but in the end, it helps keep rights straight between your development and production environments (and all environments in between) and helps avoid users who end up with god-like powers from getting one permission here and another there. The roles will be the same in all areas; the users who are associated with the roles are then different in production, test, and so on.

www.it-ebooks.info

I'll cover the following topics, which revolve around giving users permissions to use securables:

- *Grantable permissions*: This section covers the different sorts of database permissions and how to grant and revoke permission on securables.
- *Roles and schemas*: You'll learn how to use roles and schemas to grant rights efficiently to database securables.

These two topics will give you most of the information you need to know about setting up your database-level security.

Grantable Permissions

Using SQL Server security, you can easily build a security plan that prevents unwanted usage of your objects by any user. You can control rights to almost every object type, and in SQL Server, you can secure a tremendous number of object types. For our purposes here, I'll cover data-oriented security specifically, limited to the objects and the actions you can give or take away access to (see Table 9-1).

Table 9-1. *Database Objects and Permissions*

Object Type	Permission Type
Tables	`SELECT, INSERT, UPDATE, DELETE, REFERENCES`
Views	`SELECT, INSERT, UPDATE, DELETE`
Columns (view and table)	`SELECT, INSERT, UPDATE, DELETE`
Functions	`EXECUTE` (scalar) `SELECT` (table valued)
Stored procedures	`EXECUTE`

Most of these are straightforward and probably are familiar if you've done any SQL Server administration, although perhaps `REFERENCES` isn't familiar. Briefly, `SELECT` allows you to read data using a `SELECT` statement; `INSERT` allows you to add data, `UPDATE` to modify data, and `DELETE` to remove data. `EXECUTE` lets you execute coded objects, and `REFERENCES` allows objects that one user owns to reference another one via a foreign key. This is for the situation in which tables are owned by schemas that are then owned by different database principals. If you wanted to apply a foreign key between the tables, you'd be required to give the child table `REFERENCES` permissions.

As briefly mentioned earlier in this chapter for server and database permissions, you will use one of the three different statements to give or take away rights from each of your roles:

- `GRANT`: Gives the privilege to use an object
- `DENY`: Denies access to an object, regardless of whether the user has been granted the privilege from any other role
- `REVOKE`: Used to remove any `GRANT` or `DENY` permissions statements that have been applied to an object (This behaves like a delete of an applied permission.)

To see the user's rights in the database, you can use the `sys.database_permissions` catalog view. For example, use the following code to see all the rights that have been granted in the database:

```
SELECT  class_desc AS permission_type,
        object_schema_name(major_id) + '.' + OBJECT_NAME(major_id) AS object_name,
        permission_name, state_desc, USER_NAME(grantee_principal_id) AS Grantee
FROM    sys.database_permissions
```


www.it-ebooks.info

Using that query in the `master` database, you will be able to see users that have `CONNECT` rights, as well as the different stored procedures and tables that you have access to.

Controlling Access to Objects

One of the worst things that can happen in an organization is for a user to see data out of context and start a worry- or gossip-fest. Creating roles and associating users with them is a fairly easy task and is usually worth the effort. Once you've set up the security for a database using sufficient object groupings (as specific as need be, of course), management can be relatively straightforward.

Your goal is to allow the users to perform whatever tasks they need to but to prohibit any other tasks and not to let them see any data that they shouldn't. You can control access at several levels. At a high level, you might want to grant (or deny) a principal access to all invoices—in which case, you might control access at the level of a table. At a more granular level, you might want to control access to certain columns or rows within that table. In a more functional approach, you might give rights only to use stored procedures to access data. All these approaches are commonly used in the same database in some way, shape, or form.

In this section, I'll cover the built-in security for tables and columns as it's done using the built-in vanilla security, before moving on to the more complex strategies using coded objects like views and stored procedures.

Table Security

As already mentioned, for tables at an object level, you can grant a principal rights to `INSERT`, `UPDATE`, `DELETE`, or `SELECT` data from a table. This is the most basic form of security when dealing with data. The goal when using table-based security is to keep users looking at, or modifying, the entire set of data, rather than specific rows. We'll progress to the specific security types as we move through the chapter. As mentioned in the introduction to this section, all objects should be owned by the same user for most normal databases (not to be confused with the owner from the previous versions of SQL Server), so we won't deal with the `REFERENCES` permission type.

■ **Note** In the context of security, a view will be treated just like a table, in that you can grant `INSERT`, `UPDATE`, `DELETE`, and/or `SELECT` rights to the view. Views have other considerations that will be covered later in this chapter.

As an example of table security, you'll create a new table, and I'll demonstrate, through the use of a new user, what the user can and cannot do:

```
USE ClassicSecurityExample;
GO
--start with a new schema for this test and create a table for our demonstrations
CREATE SCHEMA TestPerms;
GO
CREATE TABLE TestPerms.TableExample
(
  TableExampleId int IDENTITY(1,1)
        CONSTRAINT PKTableExample PRIMARY KEY,
  Value varchar(10)
);
```

Next, create a new user, without associating it with a login. You won't need a login for many of the examples, because you'll use impersonation to pretend to be the user without logging in:

```
CREATE USER Tony WITHOUT LOGIN;
```


www.it-ebooks.info

■ **Note** The ability to have a user without login privileges allows you to have objects in the database that aren't actually owned by a particular login, making managing objects cleaner, particularly when you drop a login that was connected to a user or restore a database.

You impersonate the user Tony and try to create a new row:

```
EXECUTE AS USER = 'Tony';

INSERT INTO TestPerms.TableExample(Value)
VALUES ('a row');
```

Well, as you would (or, at least, will come to) expect, here's the result:

```
Msg 229, Level 14, State 5, Line 2
The INSERT permission was denied on the object 'TableExample', database
'ClassicSecurityExample', schema 'TestPerms'.
```

Now, go back to being the dbo using the REVERT command, give the user rights, return to being Tony, and try to insert again:

```
REVERT; --return to admin user context
GRANT INSERT ON TestPerms.TableExample TO Tony;
GO
```

Then, try to execute the insert statement again as Tony; you should now be able to execute the insert statement:

```
EXECUTE AS USER = 'Tony';

INSERT INTO TestPerms.TableExample(Value)
VALUES ('a row');
```

No errors here. Now, because Tony just created the row, the user should be able to select the row, right?

```
SELECT TableExampleId, Value
FROM   TestPerms.TableExample;
```

No, the user had rights only to INSERT data, not to view it:

```
Msg 229, Level 14, State 5, Line 1
The SELECT permission was denied on the object 'TableExample', database
'ClassicSecurityExample', schema 'TestPerms'.
```

Now, you can give the user Tony rights to SELECT data from the table using the following GRANT statement:

```
REVERT;
GRANT SELECT ON TestPerms.TableExample TO Tony;
```

Now that Tony has rights, you can successfully run the following:

```
EXECUTE AS USER = 'Tony';

SELECT TableExampleId, Value
```


www.it-ebooks.info

```
FROM   TestPerms.TableExample;

REVERT;
```

The SELECT statement works and does return the row the user created. At the table level, you can do this individually for each of the four DML statements INSERT, UPDATE, DELETE, and SELECT (or you can use GRANT ALL ON <objectName> TO <principal> to give all rights to the <objectName> to the <principal>. The goal is to give the users only what they need. For example, if the user happened to represent a device that was inserting readings, it wouldn't need to be able to read, modify, or destroy data, just create it.

Column-Level Security

For the most part, it's enough simply to limit a user's access at the level of either being able to use (or not use) the entire table or view, but as the next two major sections of the chapter will discuss, sometimes the security needs to be more granular. Sometimes you need to restrict users to using merely part of a table. In this section, I'll look at the security syntax that SQL Server provides at a basic level to grant rights at a column level. Later in this chapter, I'll present other methods that use views or stored procedures.

For our example, we'll create a couple of database users:

```
CREATE USER Employee WITHOUT LOGIN;
CREATE USER Manager WITHOUT LOGIN;
```

Then, we'll create a table to use for our column-level security examples for a Product table. This Product table has the company's products, including the current price and the cost to produce this product:

```
CREATE SCHEMA Products;
GO
CREATE TABLE Products.Product
(
        ProductId int identity CONSTRAINT PKProduct PRIMARY KEY,
        ProductCode varchar(10) CONSTRAINT AKProduct_ProductCode UNIQUE,
        Description varchar(20),
        UnitPrice decimal(10,4),
        ActualCost decimal(10,4)
);
INSERT INTO Products.Product(ProductCode, Description, UnitPrice, ActualCost)
VALUES  ('widget12','widget number 12',10.50,8.50),
        ('snurf98','Snurfulator',99.99,2.50);
```

Now, we want our employees to be able to see all the products, but we don't want them to see what each product costs to manufacture. The syntax is the same as using GRANT on a table, but we include in parentheses a comma-delimited list of the columns to which the user is being denied access. In the next code block, we grant SELECT rights to both users but take away these rights on the ActualCost column:

```
GRANT SELECT ON Products.Product TO employee, manager;
DENY SELECT ON Products.Product (ActualCost) TO employee;
```

To test our security, we impersonate the manager:

```
EXECUTE AS USER = 'manager';
SELECT  *
FROM    Products.Product;
```


www.it-ebooks.info

This returns all columns with no errors:

```
ProductId      ProductCode      Description          UnitPrice      ActualCost
---------      -----------      ----------------     ---------      ----------
1              widget12         widget number 12     10.5000        8.5000
2              snurf98          Snurfulator          99.9900        2.5000
```

■ **Tip** I know you are probably thinking that it's bad practice to use `SELECT *` in a query. It's true that using `SELECT *` in your permanent code is a bad idea, but generally speaking, when writing ad hoc queries, most users use the * shorthand for all columns, and it is perfectly acceptable to do so. It is in stored objects that the * shorthand is considered bad. It is also an essential part of the example I am presenting!

The manager worked fine; what about the employee?

```
REVERT; --revert back to SA level user or you will get an error that the
        --user cannot do this operation because the manager user doesn't
        --have rights to impersonate the employee
GO
EXECUTE AS USER = 'employee';
GO
SELECT  *
FROM    Products.Product;
```

This returns the following result:

```
Msg 230, Level 14, State 1, Line 1
The SELECT permission was denied on the column 'ActualCost' of the object 'Product', database
'ClassicSecurityExample', schema 'Products'.
```

"Why did I get this error?" the user first asks, then (and this is harder to explain), "How do I correct it?" You might try to explain to the user, "Well, just list all the columns you *do* have access to, without the columns you cannot see, like this:"

```
SELECT ProductId, ProductCode, Description, UnitPrice
FROM   Products.Product;
REVERT;
```

This returns the following results for the user `employee`:

```
ProductId      ProductCode      Description          UnitPrice
---------      -----------      ----------------     ---------
1              widget12         widget number 12     10.5000
2              snurf98          Snurfulator          99.9900
```


www.it-ebooks.info

The answer, although technically correct, isn't even vaguely what the user wants to hear. "So every time I want to build an ad hoc query on the `Product` table (which has 87 columns instead of the 5 we've generously mocked up for your learning ease), I have to type out all the columns? And I have to remember all of the columns? Ugh!"

This is why, for the most part, column-level security is rarely used as a primary security mechanism, because of how it's implemented. You don't want users getting error messages when they try to run a fairly simple query on a table. You might add column-level security to the table "just in case," but for the most part, use coded objects such as stored procedures or views to control access to certain columns. I'll discuss these solutions in the next section.

Here's one last tidbit about column security syntax: once you've applied the `DENY` option on a column, to give the user rights you need to `REVOKE` the `DENY` to restore the ability to access the column and then `GRANT` access to the entire table. Using `REVOKE` alone would only delete the `DENY`.

Roles

Core to the process of granting rights is who to grant rights to. I've introduced the database user, commonly referred to as just *user*. The user is the lowest level of security principal in the database and can be mapped to logins, certificates, and asymmetrical keys, or even not mapped to a login at all (either a user created with the `WITHOUT LOGIN` option specifically for impersonation, or they can be orphaned by dropped users). In this section, I will expand a bit more on just what a role is.

Roles are groups of users and other roles that allow you to grant object access to multiple users at once. Every user in a database is a member of at least the `public` role, which will be mentioned again in the "Built-in Database Roles" section, but may be a member of multiple roles. In fact, roles may be members of other roles. I'll discuss a couple types of roles:

- *Built-in database roles*: Roles that are provided by Microsoft as part of the system
- *User-defined database roles*: Roles, defined by you, that group Windows users together in a user-defined package of rights
- *Application roles*: Roles that are used to give an application specific rights, rather than to a group or individual user

Each of these types of roles is used to give rights to objects in a more convenient manner than granting them directly to an individual user. Many of these possible ways to implement roles (and all security really) are based on the politics of how you get to set up security in your organization. There are many different ways to get it done, and a lot of it is determined by who will do the actual work. End users may need to give another user rights to do some things, as a security team, network administrators, DBAs, and so on, also dole out rights. The whole idea of setting up roles to group users is to lower the amount of work required to get things done and managed right.

Built-in Database Roles

As part of the basic structure of the database, Microsoft provides a set of nine built-in roles that give a user a special set of rights at a database level:

- `db_owner`: Users associated with this role can perform any activity in the database.
- `db_accessadmin`: Users associated with this role can add or remove users from the database.
- `db_backupoperator`: Users associated with this role are allowed to back up the database.
- `db_datareader`: Users associated with this role are allowed to read any data in any table.

www.it-ebooks.info

- `db_datawriter`: Users associated with this role are allowed to write any data in any table.
- `db_ddladmin`: Users associated with this role are allowed to add, modify, or drop any objects in the database (in other words, execute any DDL statements).
- `db_denydatareader`: Users associated with this role are denied the ability to see any data in the database, though they may still see the data through stored procedures.
- `db_denydatawriter`: Much like the `db_denydatareader` role, users associated with this role are denied the ability to modify any data in the database, though they still may modify data through stored procedures.
- `db_securityadmin`: Users associated with this role can modify and change permissions and roles in the database.

Of particular interest in these groups to many DBAs and developers are the `db_datareader` and `db_datawriter` roles. All too often these roles (or, unfortunately, the `db_owner` role) are the only permissions ever used in the database. For most any database, this should rarely be the case. Even when the bulk of the security is being managed by the user interface, there are going to be tables that you may not want users to be able to access. As an example, in my databases, I almost always have a `utility` schema that I place objects in to implement certain database-level utility tasks. If I wanted to keep up with the counts of rows in tables on a daily basis, I would create a row in the table each day with the row count of each table. If I want a procedure to drop all of the constraints on a database for a given process, I would have a procedure in the utility schema as well. If users accidentally executes that procedure instead of the benign query procedure they were trying to click, it is your fault, not theirs.

The point is that security should be well planned out and managed in a thoughtful manner, not just managed by giving full access and hoping for the best from the user interface standpoint. As I will introduce in the "Schemas" section, instead of using the `db_datareader` fixed role, consider granting `SELECT` permissions at the schema level. If you do, any new schema added for some purpose will not automatically be accessible to everyone by the `db_datareader` membership, but all of the objects in that schema (even new ones) will automatically get the existing schema permission. My goal is to limit fixed schema use to utility users, like an ETL program's access, for example, that will not be doing any ad hoc queries that could be in error.

User-Defined Database Roles

In addition to the fixed database roles, you can create your own database roles to grant rights to database objects. To a role, you can give or deny rights to use tables and code in the database, as well as database-level rights such as `ALTER`, `ALTER ANY USER`, `DELETE` (from any table), `CREATE ROLE`, and so on. You can control rights to database management and data usage together in the same package, rather than needing to grant users ownership of the database where they have unlimited power to make your day busy restoring from backups and fixing the database.

Roles should be used to create a set of database rights for a job description, or perhaps an aspect of a job description. Take, for example, any typical human resources system that has employee information such as name, address, position, manager, pay grade, and so on. We'll likely need several roles, such as the following to cover all the common roles that individuals and some processes need to do their job:

- `Administrators`: Should be able to do any task with data, including ad hoc access; will also need rights to back up and restore the database on occasion (using a user interface, naturally).
- `HRManagers`: Should be able to do any task in the system with data.
- `HRWorkers`: Can maintain any attribute in the system, but approval rows are required to modify salary information.

www.it-ebooks.info

- `Managers`: All managers in the company might be in a role like this, which might give them view rights to high-level corporate information. You can then limit them to only the ability to see the details for their own workers, using further techniques I'll present in the section "Implementing Configurable Row-Level Security with Views" later in this chapter.
- `Employees`: Can see only their own information and can modify only their own personal address information.

Setting up a tight security system isn't an easy task, and it takes lots of thought, planning, and hard work to get it done right.

Each of the roles would then be granted access to all the resources that they need. A member of the `Managers` role would likely also be a member of the `Employees` role. That way, the managers could see the information for their employees and also for themselves. Users can be members of multiple roles, and roles can be members of other roles. Permissions are additive, so if a user is a member of three roles, the user has an effective set of permissions that's the union of all permissions of the groups, for example:

- `Managers`: Can view the `Employees` table
- `Employees`: Can view the `Product` table
- `HRWorkers`: Can see employment history

If the `Managers` role were a member of the `Employees` role, a member of the `Managers` role could do activities that were enabled by either role. If a user were a member of the `HRWorkers` group and the `Employees` role, the user could see employment history and the `Product` table (it might seem logical that users could see the `Employees` table, but this hasn't been explicitly set in our tiny example). If a manager decides that making the lives of others miserable is no longer any fun, as part of the demotion, that user would be removed from the `Managers` role.

Programmatically, you can determine some basic information about a user's security information in the database:

- `IS_MEMBER ('<role>')`: Tells you whether the current user is the member of a given role. This is useful for building security-based views.
- `USER`: Tells you the current user's name in the database.
- `HAS_PERMS_BY_NAME`: Lets you interrogate the security system to see what rights a user has. This function has a complex public interface, but it's powerful and useful.

You can use these functions in applications and T-SQL code to determine at runtime what the user can do. For example, if you wanted only `HRManager` members to execute a procedure, you could check this:

```
SELECT IS_MEMBER('HRManager');
```

A return value of `1` means the user is a member (`0` means not a member) of the role. A procedure might start out like the following:

```
IF (SELECT IS_MEMBER('HRManager')) = 0 OR (SELECT IS_MEMBER('HRManager')) IS NULL
    SELECT 'I..DON''T THINK SO!';
```

This prevents even the database owner from executing the procedure, though `dbo` users can obviously get the code for the procedure and execute it if they're desirous enough (the "Monitoring and Auditing" section of this chapter covers some security precautions to handle nosy DBA types), though this is generally a hard task to make bulletproof enough.

www.it-ebooks.info

■ **Tip** If there isn't an `HRManager` role configured, `is_member` will return `NULL`. If this is a consideration, be certain to code for it like I did in my query, or add another block of code to warn or log that the setup is invalid.

For example, in our HR system, if you wanted to remove access to the `salaryHistory` table just from the `Employees` role, you wouldn't deny access to the `Employees` role, because managers are employees also and would need to have rights to the `salaryHistory` table. To deal with this sort of change, you might have to revoke rights to the `Employees` role and then give rights to the other groups, rather than deny rights to a group that has lots of members.

As an example, consider that you have three users in the database:

```
CREATE USER Frank WITHOUT LOGIN;
CREATE USER Julie WITHOUT LOGIN;
CREATE USER Rie WITHOUT LOGIN;
```

Julie and Rie are members of the `HRWorkers` role, so we will add:

```
CREATE ROLE HRWorkers;

ALTER ROLE HRWorkers ADD MEMBER Julie;
ALTER ROLE HRWorkers ADD MEMBER Rie;
```

■ **Tip** `ALTER ROLE` is new to SQL Server 2012. It replaces `sp_addrolemember`.

Next, you have a `Payroll` schema, and in this is (at the least) an `EmployeeSalary` table:

```
CREATE SCHEMA Payroll;
GO
CREATE TABLE Payroll.EmployeeSalary
(
    EmployeeId int,
    SalaryAmount decimal(12,2)
);
GRANT SELECT ON Payroll.EmployeeSalary to HRWorkers;
```

Next, test the users:

```
EXECUTE AS USER = 'Frank';

SELECT *
FROM   Payroll.EmployeeSalary;
```

This returns the following error, because Frank isn't a member of this group:

```
Msg 229, Level 14, State 5, Line 3
The SELECT permission was denied on the object 'EmployeeSalary', database
'ClassicSecurityExample', schema 'Payroll'.
```


www.it-ebooks.info

However, change over to Julie:

```
REVERT;
EXECUTE AS USER = 'Julie';

SELECT *
FROM   Payroll.EmployeeSalary;
```

She can view the data of tables in the `Payroll` schema because she's a member of the role that was granted `SELECT` permissions to the table:

```
EmployeeId      SalaryAmount
----------      ------------
```

Roles are almost always the best way to apply security in a database. Instead of giving individual users specific rights, develop roles that match job positions. Granting rights to an individual is not necessarily bad. To keep this section reasonable, I won't extend the example to include multiple roles, but a user can be a member of many roles, and the user gets the cumulative effect of the chosen rights. So if there is an `HRManagers` role and `Julie` is a member of this group as well as the `HRWorkers` role, the rights of the two groups would effectively be `UNION`ed. The result would be the user's rights.

There's one notable exception: one `DENY` operation prevents another's `GRANT` operations from applying. Say Rie had her rights to the `EmployeeSalary` table denied:

```
REVERT;
DENY SELECT ON payroll.employeeSalary TO Rie;
```

Say she tried to select from the table:

```
EXECUTE AS USER = 'Rie';
SELECT  *
FROM    payroll.employeeSalary;
```

She would be denied:

```
Msg 229, Level 14, State 5, Line 2
The SELECT permission was denied on the object 'EmployeeSalary', database
'ClassicSecurityExample', schema 'Payroll'.
```

This denial of access is true even though she was granted rights via the `HRWorkers` group. This is why `DENY` is generally not used much. Rarely will you punish users via rights, if for no other reason than keeping up with the rights can be too difficult. You might apply `DENY` to a sensitive table or procedure to be certain it wasn't used, but only in limited cases.

If you want to know from which tables the user can `SELECT`, you can use a query such as the following while in the context of the user. Reverting to your `sysadmin` login–based user, executing this query will return the three tables we have created so far in this database. A bit more interesting is what happens when I check the permissions when the user is `Julie`:

```
REVERT ;
EXECUTE AS USER = 'Julie';

--note, this query only returns rows for tables where the user has SOME rights
SELECT TABLE_SCHEMA + '.' + TABLE_NAME AS tableName,
```


www.it-ebooks.info

```
        HAS_PERMS_BY_NAME(TABLE_SCHEMA + '.' + TABLE_NAME, 'OBJECT', 'SELECT')
                AS allowSelect,
        HAS_PERMS_BY_NAME(TABLE_SCHEMA + '.' + TABLE_NAME, 'OBJECT', 'INSERT')
                AS allowInsert
FROM INFORMATION_SCHEMA.TABLES;
```

This returns

```
tableName                   allowSelect       allowInsert
---------------------       -----------       -----------
Payroll.EmployeeSalary      1                 0
```

User `Julie` has rights to see only one of the tables we have created, and she has only select rights. Applications that use direct access to the tables can use a query such as this to determine what actions users can do and adjust the user interface to match their rights. Finally, you will need to use `REVERT` to go back to the security context of the power user to continue to the next examples.

Application Roles

Developers commonly like to set up applications using a single login and then manage security in the application. This can be an adequate way to implement security, but it requires you to re-create all the login stuff, when you could use simple Windows Authentication to check whether a user can execute an application. Application roles let you use the SQL Server login facilities to manage who a person is and if that person has rights to the database and then let the application perform the finer points of security.

To be honest, this can be a nice mix, because the hardest part of implementing security isn't restricting a person's ability to do an activity; it's nicely letting them know by hiding actions they cannot do. I've shown you a few of the security catalog views already, and there are more in Books Online. Using them, you can query the database to see what a user can do to help facilitate this process. However, it isn't a trivial task and is often considered too much trouble, especially for homegrown applications.

An application role is almost analogous to using `EXECUTE AS` to set rights to another user, but instead of a person, the user is an application. You change to the context of the application role using `sp_setapprole`. You grant the application role permissions just like any other role, by using the `GRANT` statement.

As an example of using an application role, you'll create both a user named `Bob` and an application role and give them totally different rights. The `TestPerms` schema was created earlier, so if you didn't create it before, go ahead and do so.

```
CREATE TABLE TestPerms.BobCan
(
    BobCanId int identity(1,1) CONSTRAINT PKBobCan PRIMARY KEY,
    Value varchar(10)
);
CREATE TABLE TestPerms.AppCan
(
    AppCanId int identity(1,1) CONSTRAINT PKAppCan PRIMARY KEY,
    Value varchar(10)
);
```

Now, create the user `Bob` to correspond to the `BobCan` table:

```
CREATE USER Bob WITHOUT LOGIN;
```


www.it-ebooks.info

Next, give Bob SELECT rights to his table:

```
GRANT SELECT ON TestPerms.BobCan TO Bob;
GO
```

Finally, create an application role, and give it rights to its table:

```
CREATE APPLICATION ROLE AppCan_application WITH PASSWORD = '39292LjAsll2$3';
GO
GRANT SELECT ON TestPerms.AppCan TO AppCan_application;
```

You will probably note that one of the drawbacks to using an application role is that it requires a password. This password is passed in clear text to the SQL Server, so make sure that, first, the password is complex and, second, you encrypt any connections that might be using these when there's a threat of impropriety. There is an encryption option that will obfuscate the password, but it is only available with an ODBC or OleDB client. For more security, use an encrypted connection. For more information on encrypting connections, look up "Secure Sockets Layer (SSL)" in Books Online.

Next, set the user you're working as to Bob:

```
EXECUTE AS USER = 'Bob';
```

Now, try to retrieve data from the BobCan table:

```
SELECT * FROM TestPerms.BobCan;
```

It works with no error:

```
BobCanId        Value
--------        -----
```

However, try retrieving data from the AppCan table:

```
SELECT * FROM TestPerms.AppCan;
```

The following error is returned:

```
Msg 229, Level 14, State 5, Line 1
The SELECT permission was denied on the object 'AppCan', database 'ClassicSecurityExample',
schema 'TestPerms'.
```

This isn't surprising, because Bob has no permissions on the AppCan table. Next, still logged in as Bob, use the sp_setapprole procedure to change the security context of the user to the application role, and the security is reversed:

```
REVERT;
GO
EXECUTE sp_setapprole 'AppCan_application', '39292LjAsll2$3';
GO
SELECT * FROM TestPerms.BobCan;
```

This returns the following error:

www.it-ebooks.info

```
Msg 229, Level 14, State 5, Line 1
The SELECT permission was denied on the object 'BobCan', database 'ClassicSecurityExample',
schema 'TestPerms'.
```

That's because you're now in context of the application role, and the application role doesn't have rights to the table. Finally, the application role can read from the AppCan table:

```
SELECT * FROM TestPerms.AppCan;
```

This doesn't return an error:

```
AppCanId        Value
--------        -----
```

When you're in the application role context, you look to the database as if you're the application, not your user, as evidenced by the following code:

```
SELECT USER AS userName, SYSTEM_USER AS login;
```

This returns the following result:

```
userName               login
------------------     --------------------
AppCan_application     DENALI-PC\AlienDrsql
```

The login returns whatever login name you're logged in as, without regard to any impersonation, because the user is at the database level and the login is at the server level. Once you've executed sp_setapprole, the security stays as this role until you disconnect from the SQL Sever server or execute sp_unsetapprole. However, sp_unsetapprole doesn't work nearly as elegantly as REVERT, because you need to have a "cookie" value stored to be able to go back to your previous database security context.

■ **Note** You will need to disconnect and reconnect at this point, because you will be stuck in the application role state.

To demonstrate, log back in as your sysadmin role user:

```
--Note that this must be executed as a single batch because of the variable
--for the cookie
DECLARE @cookie varbinary(8000);
EXECUTE sp_setapprole 'AppCan_application', '39292LjAsll2$3'
        , @fCreateCookie = true, @cookie = @cookie OUTPUT;

SELECT @cookie AS cookie;
SELECT USER AS beforeUnsetApprole;

EXEC sp_unsetapprole @cookie;

SELECT USER AS afterUnsetApprole;

REVERT; --done with this user
```


www.it-ebooks.info

This returns the following results:

```
Cookie
------------------------------------------------------------------------------------------
0x39881A28E9FB46A0A002ABA31C11B7F4C149D8CB2BCF99B7863FFF729E2BE48F13C0F83BAD62CF0B221A863B83

beforeUnsetApprole
------------------
AppCan_application

afterUnsetApprole
-----------------
dbo
```

The cookie is an interesting value, much larger than a GUID—it was declared as `varbinary(8000)` in Books Online, so I used that data type as well. It does change for each execution of the batch. The fact is, it is fairly unlikely to want to unset the application role for most usages.

Schemas

Schemas were introduced and used heavily in the previous chapters, and up to this point, they've been used merely as a method to group like objects. Logical grouping is an important usage of schemas, but it is only one of they uses. Using these logical groups to apply security is where they really pay off. A user owns a schema, and a user can also own multiple schemas. For most any database that you'll develop for a system, the best practice is to let all schemas be owned by the `dbo` system user. You might remember from versions before 2005 that `dbo` owned all objects, and although this hasn't technically changed, it is the schema that is owned by `dbo`, and the table in the schema. Hence, instead of the reasonably useless `dbo` prefix being attached to all objects representing the owner, you can nicely group together objects of a common higher purpose and then (because this is a security chapter) grant rights to users at a schema level, rather than at an individual object level.

For our database-design purposes, we will assign rights for users to use the following:

- Tables and (seldomly) individual columns
- Views
- Synonyms (which can represent any of these things and more)
- Functions
- Procedures

You can grant rights to other types of objects, including user-defined aggregates, queues, and XML schema collections, but I won't cover them here. As an example, in the `AdventureWorks2012` database, there's a `HumanResources` schema. Use the following query of the `sys.objects` catalog view (which reflects schema-scoped objects):

```
USE AdventureWorks2012;
GO
SELECT type_desc, count(*)
FROM   sys.objects
WHERE  schema_name(schema_id) = 'HumanResources'
```


www.it-ebooks.info

```
  AND  type_desc IN ('SQL_STORED_PROCEDURE','CLR_STORED_PROCEDURE',
                     'SQL_SCALAR_FUNCTION','CLR_SCALAR_FUNCTION',
                     'CLR_TABLE_VALUED_FUNCTION','SYNONYM',
                     'SQL_INLINE_TABLE_VALUED_FUNCTION',
                     'SQL_TABLE_VALUED_FUNCTION','USER_TABLE','VIEW')
GROUP BY type_desc;
GO
USE ClassicSecurityExample;
```

This query shows how many of each object can be found in the version of the `HumanResources` schema I have on my laptop. As mentioned previously in this chapter, to grant privileges to a schema to a role or user, you prefix the schema name with `SCHEMA::` to indicate the type of object you are granting to. To give the users full usage rights to all these, you can use the following command:

```
GRANT EXECUTE, SELECT, INSERT, UPDATE, DELETE ON
                       SCHEMA::<schemaname> TO <database_principal>;
```

By using schemas and roles liberally, the complexity of granting rights to users on database objects can be pretty straightforward. That's because, instead of having to make sure rights are granted to 10 or even 100 stored procedures to support your application's `Customer` section, you need just a single line of code:

```
GRANT EXECUTE on SCHEMA::Customer TO CustomerSupport;
```

Bam! Every user in the `CustomerSupport` role now has access to the stored procedures in this schema. Nicer still is that even new objects added to the schema at a later date will be automatically accessible to people with rights at the schema level. For example, create a user named `Tom`; then, grant `Tom` `SELECT` rights on the `TestPerms` schema created in a previous section:

```
USE ClassicSecurityExample;
GO
CREATE USER Tom WITHOUT LOGIN;
GRANT SELECT ON SCHEMA::TestPerms TO Tom;
```

Immediately, Tom has rights to select from the tables that have been created:

```
EXECUTE AS USER = 'Tom';
GO
SELECT  * FROM TestPerms.AppCan;
GO
REVERT;
```

But also, Tom gets rights to the new table that we create here:

```
CREATE TABLE TestPerms.SchemaGrant
(
    SchemaGrantId int PRIMARY KEY
);
GO
EXECUTE AS USER = 'Tom';
GO
SELECT * FROM TestPerms.schemaGrant;
GO
REVERT;
```

Essentially, a statement like `GRANT SELECT ON SCHEMA::` is a much better way to give a user read rights to the database than using the `db_datareader` fixed database role, especially if you use schemas. This ensures that if a

www.it-ebooks.info

new schema is created and some users shouldn't have access, they will not automatically get access, but it also ensures that users get access to all new tables that they should get.

Controlling Access to Data via T-SQL–Coded Objects

Just using the database-level security in SQL Server allows you to give a user rights to access only certain objects, but as you've seen, the database-level security doesn't work in an altogether user-friendly manner, nor does it give you a great amount of specific control. You can control access to the entire table or, at the most, restrict access at a column level. In many cases, you might want to let a user join to a table to get a value but not to browse the entire table using a `SELECT` statement. Using table/object-level security alone, this is impossible, but using T-SQL coded objects, it is very possible.

Now, we get down to the business of taking complete control over database access by using the following types of objects:

- *Stored procedures and scalar functions*: These objects give users an API to the database, and then, the DBA can control security based on what the procedure does.
- *Views and table-valued functions*: In cases where the tools being used can't use stored procedures, you can still use views to present an interface to the data that appears to the user as a normal table would. In terms of security, views and table-valued functions can be used for partitioning data vertically by hiding columns or even horizontally by providing row-level security.

Coded objects let you take control of the data in ways that not only give you security over the data from a visibility or modifiability standpoint but let you control everything the user can do. (No, "modifiability" is probably not technically a word, but it will be if you just start using it. Then Webster's will cite this book as the origin, and I will sell a million copies to curious English professors! Yeah, that'll work . . .)

Controlling security with coded objects requires an understanding of how ownership affects rights to objects. For example, if a user owns a stored procedure and that stored procedure uses other objects it owns, the user who executes the procedure doesn't need direct rights to the other objects. The name for the way rights are allowed on owned objects in coded objects is *ownership chaining*.

Just because a user can use a stored procedure or function doesn't necessarily mean that he or she will need to have rights to every object to which the stored procedure refers. As long as the owner or the object owns all the schemas for all the objects that are referenced, the ownership chain isn't broken, and any user granted rights to use the object can see any referenced data. If you break the ownership chain and reference data in a schema not owned by the same user, the user will require rights granted directly to the object, instead of the object being created. This concept of the ownership chain is at the heart of why controlling object access via coded objects is so nice.

I put stored procedures along with views and functions together in a section because whichever option you choose, you will still have accomplished the separation of interface from implementation. As long as the contract between the stored procedure or view is what the developer or application is coding or being coded to, the decision of which option to select will offer different sorts of benefits.

Stored Procedures and Scalar Functions

Security in stored procedures and functions is always at the object level. Using stored procedures and functions to apply security is quite nice because you can give the user rights to do many operations without the user having rights to do the same operations on their own (or even knowing how it's done.)

In some companies, stored procedures are used as the primary security mechanism, by requiring that all access to the server be done without executing a single "raw" DML statement against the tables. By building code that encapsulates all functionality, you then can apply permissions to the stored procedures to restrict what the user can do.

www.it-ebooks.info

In security terms only, this allows you to have *situational control* on access to a table. This means that you might have two different procedures that functionally do the same operation, but giving a user rights to one procedure doesn't imply that he or she has rights to the other. (I will discuss more about the pros and cons of different access methods in Chapter 13, but in this chapter, we will to limit our discussion to the security aspects.)

Take, for example, the case where a screen is built using one procedure; the user might be able to do an action, such as deleting a row from a specific table. But when the user goes to a different screen that allows deleting 100 rows, that ability might be denied. What makes this even nicer is that with decent naming of your objects, you can give end users or managers rights to dole out security based on actions they want their employees to have, without needing the IT staff to handle it.

As an example, create a new user for the demonstration:

```
CREATE USER procUser WITHOUT LOGIN;
```

Then (as `dbo`), create a new schema and table:

```
CREATE SCHEMA procTest;
GO
CREATE TABLE procTest.misc
(
    Value varchar(20),
    Value2 varchar(20)
);
GO
INSERT  INTO procTest.misc
VALUES  ('somevalue','secret'),
        ('anothervalue','secret');
```

Next, we will create a stored procedure to return the values from the `value` column in the table, not the `value2` column; then, we grant rights to the `procUser` to execute the procedure:

```
CREATE PROCEDURE procTest.misc$select
AS
    SELECT Value
    FROM procTest.misc;
GO
GRANT EXECUTE on procTest.misc$select to procUser;
```

After that, change the context to the `procUser` user and try to `SELECT` from the table:

```
EXECUTE AS USER = 'procUser';
GO
SELECT Value, Value2
FROM   procTest.misc;
```

You get the following error message, because the user hasn't been given rights to access this table:

```
Msg 229, Level 14, State 5, Line 1
The SELECT permission was denied on the object 'misc', database 'ClassicSecurityExample',
schema 'procTest'.
```

However, execute the following procedure:

```
EXECUTE procTest.misc$select;
```


www.it-ebooks.info

The user does have access to execute the procedure, so you get the results expected:

```
Value
------------
somevalue
anothervalue
```

This is the best way to architect a database solution. It leaves a manageable surface area, gives you a lot of control over what SQL is executed in the database, and lets you control data security nicely.

You can see what kinds of access a user has to stored procedures by executing the following statement:

```
SELECT schema_name(schema_id) +'.' + name AS procedure_name
FROM   sys.procedures;

REVERT;
```

While in the context of the `procUser`, you will see the one row for the `procTest.misc$select` procedure returned. If you were using only stored procedures to access the data, this query could be executed by the application programmer to know everything the user can do in the database.

■ **Tip** If you don't like using stored procedures as your access layer, I know you can probably make a list of reasons why you disagree with this practice. However, as I mentioned, this is largely considered a best practice in the SQL Server architect community because of not only the security aspects of stored procedures but also the basic encapsulation reasons I will discuss in Chapter 13. A lot of applications using object relational mapping layers will not work with stored procedures, at least not working in "easy" mode, which would mean a noticeable drop off in coding performance, leading to unhappy managers, no matter what the future benefit may be.

Impersonation within Objects

I already talked about the `EXECUTE AS` statement, and it has some great applications, but using the `WITH EXECUTE` clause on a procedure or function declaration can give you some incredible flexibility to give the executor greater powers than might have been possible otherwise, certainly not without granting additional rights. Instead of changing context before an operation, you can change context while executing a stored procedure, function, or DML trigger (plus queues for Service Broker, but I won't be covering that topic). Unfortunately, the `WITH EXECUTE` clause is not available for views, because they are not technically executable objects (hence the reason why you grant `SELECT` rights and not `EXECUTE` ones).

By adding the following code, you can change the security context of a procedure to a different server or database principal when the execution begins:

```
CREATE PROCEDURE <schemaName>.<procedureName>
WITH EXECUTE AS <'loginName' | CALLER | SELF | OWNER>;
```

The different options for whom to execute as are as follows:

- `'userName'`: A specific user principal in the database.
- `CALLER`: The context of the user who called the procedure. This is the default security context you get when executing an object.

www.it-ebooks.info

- SELF: It's in the context of the user who created the procedure.
- OWNER: It's executed in the context of the owner of the module or schema.

Note that using EXECUTE AS doesn't affect the ownership chaining of the call. The security of the statements in the object is still based on the security of the schema owner. Only when the ownership chain is broken will the ownership chaining come into play. The following statements go along with the EXECUTE AS clause:

- EXECUTE AS CALLER: You can execute this in your code to go back to the default, where access is as the user who actually executed the object.
- REVERT: This reverts security to the security specified in the WITH EXECUTE AS clause.

As an example, I'll show how to build a situation where one schema owner has a table and where the next schema owner has a table and a procedure that the schema owner wants to use to access the first user's table. Finally, you have an average user who wants to execute the stored procedure.

First, you create a few users and give them rights to create objects in the database. The three users are named as follows:

- schemaOwner: This user owns the primary schema where one of the objects resides.
- procedureOwner: This user owns the owner of an object and a stored procedure.
- aveSchlub: This is the average user who finally wants to use procedureOwner's stored procedure.

So, now create these users and grant them rights:

```
--this will be the owner of the primary schema
CREATE USER schemaOwner WITHOUT LOGIN;
GRANT CREATE SCHEMA TO schemaOwner;
GRANT CREATE TABLE TO schemaOwner;

--this will be the procedure creator
CREATE USER procedureOwner WITHOUT LOGIN;
GRANT CREATE SCHEMA TO procedureOwner;
GRANT CREATE PROCEDURE TO procedureOwner;
GRANT CREATE TABLE TO procedureOwner;
GO
--this will be the average user who needs to access data
CREATE USER aveSchlub WITHOUT LOGIN;
```

Then, you change to the context of the main object owner, create a new schema, and create a table with some rows:

```
EXECUTE AS USER = 'schemaOwner';
GO
CREATE SCHEMA schemaOwnersSchema;
GO
CREATE TABLE schemaOwnersSchema.Person
(
        PersonId  int constraint PKtestAccess_Person PRIMARY KEY,
        FirstName varchar(20),
        LastName  varchar(20)
);
```


www.it-ebooks.info

```
GO
INSERT  INTO schemaOwnersSchema.Person
VALUES  (1, 'Phil','Mutayblin'),
        (2, 'Del','Eets');
```

Next, this user gives SELECT permissions to the procedureOwner user:

```
GRANT SELECT ON schemaOwnersSchema.Person TO procedureOwner;
```

After that, you set context to the secondary user to create the procedure:

```
REVERT  --we can step back on the stack of principals,
        --but we can't change directly
        --to procedureOwner. Here I step back to the db_owner user you have
        --used throughout the chapter
GO
EXECUTE AS USER = 'procedureOwner';
```

Then, you create a schema and another table, owned by the procedureOwner user, and add some simple data for the demonstration:

```
CREATE SCHEMA procedureOwnerSchema;
GO
CREATE TABLE procedureOwnerSchema.OtherPerson
(
        personId  int constraint PKtestAccess_person PRIMARY KEY,
        FirstName varchar(20),
        LastName  varchar(20)
);
GO
INSERT INTO procedureOwnerSchema.OtherPerson
VALUES (1, 'DB','Smith');
INSERT INTO procedureOwnerSchema.OtherPerson
VALUES (2, 'Dee','Leater');
```

You can see the owners of the objects and their schema using the following query of the catalog views:

```
REVERT;

SELECT tables.name AS [table], schemas.name AS [schema],
        database_principals.name AS [owner]
FROM    sys.tables
          JOIN sys.schemas
            ON tables.schema_id = schemas.schema_id
          JOIN sys.database_principals
            ON database_principals.principal_id = schemas.principal_id
WHERE tables.name IN ('Person','OtherPerson');
```

This returns the following:

```
table          schema                  owner
-----------    --------------------    --------------
OtherPerson    procedureOwnerSchema    procedureOwner
Person         schemaOwnersSchema      schemaOwner
```


www.it-ebooks.info

Next, you create two procedures as the procedureOwner user, one for the WITH EXECUTE AS as CALLER, which is the default, and then SELF, which puts it in the context of the creator, in this case procedureOwner:

```
EXECUTE AS USER = 'procedureOwner';
GO

CREATE PROCEDURE procedureOwnerSchema.person$asCaller
WITH EXECUTE AS CALLER --this is the default
AS
BEGIN
  SELECT personId, FirstName, LastName
  FROM   procedureOwnerSchema.OtherPerson; --<-- ownership same as proc

  SELECT personId, FirstName, LastName
  FROM   schemaOwnersSchema.person; --<-- breaks ownership chain
END;
GO

CREATE PROCEDURE procedureOwnerSchema.person$asSelf
WITH EXECUTE AS SELF --now this runs in context of procedureOwner,
                     --since it created it
AS
BEGIN
  SELECT personId, FirstName, LastName
  FROM   procedureOwnerSchema.OtherPerson; --<-- ownership same as proc

  SELECT personId, FirstName, LastName
  FROM   schemaOwnersSchema.person; --<-- breaks ownership chain
END;
```

Next, you grant rights on the proc to the aveSchlub user:

```
GRANT EXECUTE ON procedureOwnerSchema.person$asCaller TO aveSchlub;
GRANT EXECUTE ON procedureOwnerSchema.person$asSelf TO aveSchlub;
```

Then, you change to the context of the aveSchlub:

```
REVERT; EXECUTE AS USER = 'aveSchlub';
```

Finally, you execute the procedure:

```
--this proc is in context of the caller, in this case, aveSchlub
EXECUTE procedureOwnerSchema.person$asCaller;
```

This produces the following output, because the ownership chain is fine for the procedureOwnerSchema object, but not for the schemaOwnerSchema:

```
personId      FirstName         LastName
--------      ---------         --------
1             DB                Smith
2             Dee               Leater

Msg 229, Level 14, State 5, Procedure person$asCaller, Line 7
The SELECT permission was denied on the object 'Person', database 'ClassicSecurityExample',
schema 'schemaOwnersSchema'.
```


www.it-ebooks.info

Next, you execute the asSelf variant:

```
--procedureOwner, so it works
EXECUTE procedureOwnerSchema.person$asSelf;
```

This returns two result sets:

```
personId    FirstName      LastName
--------    ---------      --------
1           DB             Smith
2           Dee            Leater

personId    FirstName      LastName
--------    ---------      --------
1           Phil           Mutayblin
2           Del            Eets
```

What makes this different is that when the ownership chain is broken, the security context you're in is the secondaryUser, not the context of the caller, aveSchlub. Using EXECUTE AS to change security context is a cool feature. Now, you can give users temporary rights that won't even be apparent to them and won't require granting any permissions.

However, EXECUTE AS isn't a feature that should be overused, and its use should definitely be monitored by code reviews! It can be all too easy just to build your procedures in the context of the dbo and forget about decent security altogether. And that is the "nice" reason for taking care in using the feature. Another reason to take care is that a malicious programmer could (if they were devious or stupid) include dangerous code that would run as if it were the database owner, which could certainly cause undesired effects.

For example, using impersonation is a great way to implement dynamic SQL calls (I will discuss more in Chapter 13 when I discuss code-level design), but if you aren't careful to secure your code against an injection attack, the attack might just be in the context of the database owner rather than the basic application user that *should* have only limited rights if you have listened to anything I have said in the rest of this chapter.

One thing that you can do with this EXECUTE AS technique is to give a user super rights temporarily in a database. For example, consider the following procedure:

```
REVERT;
GO
CREATE PROCEDURE dbo.testDboRights
AS
  BEGIN
    CREATE TABLE dbo.test
    (
        testId int
    );
  END;
```

This procedure isn't executable by any users other than one who has db_owner level rights in the database, even if they have rights to execute the procedure. Say we have the following user and give him rights (presuming "Leroy" is a male name and not just some horrible naming humor that a female had to live with) to execute the procedure:

```
CREATE USER leroy WITHOUT LOGIN;
GRANT EXECUTE ON dbo.testDboRights TO Leroy;
```


www.it-ebooks.info

Note that you grant *only* rights to the dbo.testDboRights procedure. The user leroy can execute the one stored procedure in the database:

```
EXECUTE AS USER = 'leroy';
EXECUTE dbo.testDboRights;
```

The result is as follows, because creating a table is a database permission that leroy doesn't have, either explicitly granted or as a member of a role db_owner's role:

```
Msg 262, Level 14, State 1, Procedure testDboRights, Line 5
CREATE TABLE permission denied in database 'ClassicSecurityExample'.
```

■ **Tip** If I had a nickel for every time I have seen security issues come up when new changes were moved to a production server, I would be rich. In fact, if I just had the nickels from my own mistakes, I wouldn't exactly die a pauper.

If you alter the procedure with EXECUTE AS 'dbo', the result is that the table is created, if there isn't already a table with that name that someone else has created (like if the dbo has executed this procedure previously):

```
REVERT;
GO
ALTER PROCEDURE dbo.testDboRights
WITH EXECUTE AS 'dbo'
AS
  BEGIN
    CREATE TABLE dbo.test
    (
        testId int
    );
  END;
```

Now, you can execute this procedure and have it create the table. Run the procedure twice, and you will get an error about already having a table called dbo.test in the database. For more detailed information about EXECUTE AS, check the "EXECUTE AS" topic in Books Online.

■ **Tip** As will be discussed in the cross-database security section later in this chapter, to use external resources (like a table in a different database) using impersonation, you need to set TRUSTWORTHY to ON using the ALTER DATABASE command.

In Chapter 13, a rather large section discusses the value of ad hoc SQL versus stored procedures, and we use the EXECUTE AS functionality to provide security when it comes to executing dynamic SQL in stored procedures. This is a great new feature that will bring stored procedure development—including stored procedures that are CLR-based—to new heights, because dynamic SQL-based stored procedures were a security issue in earlier versions of SQL Server.

www.it-ebooks.info

Views and Table-Valued Functions

In this section, I'll talk about using views and table-valued functions to encapsulate the views of the data in ways that leave the data in table-like structures. You might use views and table-valued functions in concert with, or in lieu of, a full stored procedure approach to application architecture. Views, as discussed in previous chapters, allow you to form pseudotables from other table sources, sometimes by adding tables together and sometimes by splitting a table up into smaller chunks. In this section, the goal is to "hide" data, like a column, from users or hide certain rows in a table, providing data security by keeping the data out of the view of the user in question.

For an overall architecture, I will always suggest that you should use stored procedures to access data in your applications, if for no other reason than you can encapsulate many data-oriented tasks in the background, giving you easy access to tweak the performance of an activity with practically zero impact to the application. (In Chapter 13, I will give some basic coverage of application architecture.)

In this section, I'll look at the following:

- *General usage*: I'll cover basic use of views to implement security.
- *Configurable row-level security*: You can use views to implement security at the row level, and this can be extended to provide user-manageable security.

Instead of the more up-front programming-heavy stored procedure methods from the previous section, simply accessing views and table-valued functions allows for more straightforward usage. Sometimes, because of the oft-repeated mantra of "just get it done," the concept of stored procedures is an impossible sale. You will lose a bit of control, and certain concepts won't be available to you (for example, you cannot use `EXECUTE AS` in a view definition), but it will be better in some cases when you don't want to dole out access to the tables directly.

General Usage

We'll use two properties of views to build a more secure database. The first is assigning privileges to users such that they can use a view, though not the underlying tables. For example, let's go back to the `Products.Product` table used earlier in this chapter. As a reminder, execute this statement (after executing `REVERT`, if you haven't already, from the previous example):

```
SELECT  *
FROM    Products.Product;
```

The following data is returned:

```
ProductId      ProductCode      Description            UnitPrice        ActualCost
---------      -----------      ----------------       ---------        ----------
1              widget12         widget number 12       10.5000          8.5000
2              snurf98          Snurfulator            99.9900          2.5000
```

We could construct a view on this:

```
CREATE VIEW Products.allProducts
AS
SELECT ProductId,ProductCode, Description, UnitPrice, ActualCost
FROM   Products.Product;
```

Selecting data from either the table or the view returns the same data. However, they're two separate structures to which you can separately assign access privileges, and you can deal with each separately. If you need to tweak the table, you might not have to modify the view. Of course, in practice, the view won't usually

www.it-ebooks.info

include the same columns and rows as the base table, but as an example, it is interesting to realize that if you build the view in this manner, there would be little, if any, difference with using the two objects, other than how the security was set up.

One of the most important things that make views useful as a security mechanism is the ability to partition a table structure, by limiting the rows or columns visible to the user. First, you'll look at using views to implement *column-level security*, which is also known as *projection* or *vertical partitioning* of the data, because you'll be dividing the view's columns. (In the next section, I will cover horizontal partitioning, or row-level security.) For example, consider that the users in a `warehouseUsers` role need only to see a list of products, not how much they cost and certainly not how much they cost to produce. You might create a view like the following to partition the columns accordingly:

```
CREATE VIEW Products.WarehouseProducts
AS
SELECT ProductId,ProductCode, Description
FROM   Products.Product;
```

By the same token, you can use table-valued functions in much the same way, though you can do more using them, including forcing some form of filter on the results. For example, you might code the following function to list all products that are less than some price:

```
CREATE FUNCTION Products.ProductsLessThanPrice
(
  @UnitPrice decimal(10,4)
)
RETURNS table
AS
  RETURN ( SELECT ProductId, ProductCode, Description, UnitPrice
           FROM Products.Product
           WHERE UnitPrice <= @UnitPrice);
```

This can be executed like the following:

```
SELECT * FROM Products.ProductsLessThanPrice(20);
```

This returns the following result:

```
ProductId      ProductCode       Description           UnitPrice
---------      -----------       ----------------      ---------
1              widget12          widget number 12      10.5000
```

Using a multistatement table-valued function (say that three times fast), you could force the condition that only users of a given security group can look at items with a price greater than some amount. Here's an example of such a function:

```
CREATE FUNCTION Products.ProductsLessThanPrice_GroupEnforced
(
  @UnitPrice decimal(10,4)
)
RETURNS @output TABLE (ProductId int,
                       ProductCode varchar(10),
                       Description varchar(20),
                       UnitPrice decimal(10,4))
```


www.it-ebooks.info

```
AS
  BEGIN
    --cannot raise an error, so you have to implement your own
    --signal, or perhaps simply return no data.
    IF @UnitPrice > 100 AND (
                        IS_MEMBER('HighPriceProductViewer') = 0
                        OR IS_MEMBER('HighPriceProductViewer') IS NULL)
    INSERT @output
    SELECT -1,'ERROR','',-1;
  ELSE
    INSERT @output (ProductId, ProductCode, Description, UnitPrice)
    SELECT ProductId, ProductCode, Description, UnitPrice
    FROM   Products.Product
    WHERE  UnitPrice <= @UnitPrice;
  RETURN;
END;
```

To test, I will create a couple of roles and add users (with what I hope are obvious enough names):

```
CREATE ROLE HighPriceProductViewer;
CREATE ROLE ProductViewer;
GO
CREATE USER HighGuy WITHOUT LOGIN;
CREATE USER LowGuy WITHOUT LOGIN;
GO
ALTER ROLE HighPriceProductViewer ADD MEMBER HighGuy;
ALTER ROLE ProductViewer ADD MEMBER HighGuy;
ALTER ROLE ProductViewer ADD MEMBER LowGuy;
GO
```

Then, I will grant rights to the function to the `ProductViewer` group *only*. This gives members of this group rights to execute the procedure, but the checks in the function still have to be passed.

```
GRANT SELECT ON Products.ProductsLessThanPrice_GroupEnforced TO ProductViewer;
```

Then, executing as the high-limit user, look for products up to $10,000:

```
EXECUTE AS USER = 'HighGuy';

SELECT *
FROM   Products.ProductsLessThanPrice_GroupEnforced(10000);

REVERT;
```

You get these results:

```
ProductId     ProductCode     Description          UnitPrice
---------     -----------     ----------------     ---------
1             widget12        widget number 12     10.5000
2             snurf98         Snurfulator          99.9900
```

But execute as the low-limit user with too large of a parameter value:

```
EXECUTE AS USER = 'LowGuy';
```


www.it-ebooks.info

```
SELECT *
FROM   Products.ProductsLessThanPrice_GroupEnforced(10000);

REVERT;
```

and you get the following error output:

```
ProductId   ProductCode   Description   UnitPrice
---------   -----------   -----------   ---------
-1          ERROR                       -1.0000
```

■ **Note** The lack of error reporting from table-valued functions is annoying at times. I chose to output some form of error description to give the example some flashiness, but just returning no values could be an acceptable output too.

Using the same GRANT syntax as in the "Table Security" section, you can give a user rights to use the view for SELECT, INSERT, UPDATE, or DELETE to a view such that it will look and act to the user just like a table. If the view is a view of multiple tables, the view might not support modifications or deletions, but you can implement INSTEAD OF triggers to allow these operations on a view to do nearly anything you need (triggers were used in several previous chapters and are covered in some detail in Appendix B).

What makes this grand is that if you aren't able to use stored procedures, because of some technical or political reason (or personal choice, I suppose), you can do most of the things that you need to do in code using INSTEAD OF triggers or, at worst, user-defined functions, and the client programmers needn't know that they exist. The only concern here is that if you change any data that the client might have cached, you might have to work this out so the data and cached copies aren't significantly out of sync.

Implementing Configurable Row-Level Security with Views

I've covered vertical partitioning, which is pretty easy. Row-level security, or *horizontally partitioning* data, isn't quite so elegant, especially if you can't use stored procedures in your applications. Using stored procedures, you could have a procedure that fixes certain operations, such as modifying active customers, or just certain products of a specific type, and so on.

With some planning, the same kind of partitioning can be done with views, and you could implement views that included all of a given type of product, another view for a different type, and yet another for other types. This scheme can work, but it isn't altogether flexible, and it's generally unnatural for a UI to have to view different objects to do ostensibly the same operation, just with a slightly different filter of data based on the security context of a user. It makes the objects tightly coupled with the data in the table. For some fixed domain tables that never change, this isn't a problem. But for many situations, users don't want to have to go back to the programming staff and ask for an implementation for which they have to jump through hoops, because the change will cost money for planning, programming, testing, and such.

In this section, I'll demonstrate a way to implement runtime configurable row-level security—using views. Views let you cut the table in sections that include all the columns, but not all the rows, based on some criteria. To the example table, you're going to add a productType column that you'll use to partition on:

```
ALTER TABLE Products.Product
  ADD   ProductType varchar(20) NULL;
GO
UPDATE  Products.Product
```


www.it-ebooks.info

```
SET     ProductType = 'widget'
WHERE   ProductCode = 'widget12';
GO
UPDATE  Products.Product
SET     ProductType = 'snurf'
WHERE   ProductCode = 'snurf98';
```

Looking at the data in the table, you can see the following results:

```
ProductId     ProductCode     Description           UnitPrice     ActualCost     ProductType
---------     -----------     ----------------      ---------     ----------     -----------
1             widget12        widget number 12      10.5000       8.5000         widget
2             snurf98         Snurfulator           99.9900       2.5000         snurf
```

As discussed, the simplest version of row-level security is just building views to partition the data. For example, suppose you want to share the widgets only with a certain group in a company, mapped to a role. You can build the following view:

```
CREATE  VIEW Products.WidgetProducts
AS
SELECT  ProductId, ProductCode, Description, UnitPrice, ActualCost
FROM    Products.Product
WHERE   ProductType = 'widget'
WITH    CHECK OPTION; --This prevents the user from entering data that would not
                      --match the view's criteria
```

Now, you can select data from this table and the user never needs to know that other products exist:

```
SELECT *
FROM   Products.WidgetProducts;
```

This returns the following result:

```
ProductId      ProductCode     Description           UnitPrice     ActualCost     ProductType
---------      -----------     ----------------      ---------     ----------     -----------
1              widget12        widget number 12      10.5000       8.5000         widget
```

You can grant `INSERT`, `UPDATE`, and `DELETE` rights to the user to modify the view as well, because it's based on one table and we set the `WITH CHECK OPTION`. This option ensures that the rows after modification remain visible through the view after the change, or in this case, that the user couldn't change the `ProductType` if it were in the `SELECT` list. The only rows a user of this view would be able to modify would be the ones where `ProductType = 'widget'`. Using `INSTEAD OF`, triggers you can code almost any security you want to for modifications on your views. For simple partitioning needs, the `CHECK OPTION` might work fine; otherwise, use triggers or stored procedures as needed.

This view can then have permissions granted to let only certain people use it. This is a decent technique when you have an easily described set, or possibly few types to work with, but can become a maintenance headache.

In the next step, ramping up row-level security, you build the following view to let users see snurfs only if they're members of the `snurfViewer` role, using the `is_member` function:

```
CREATE  VIEW Products.ProductsSelective
AS
SELECT  ProductId, ProductCode, Description, UnitPrice, ActualCost
```


www.it-ebooks.info

```
FROM    Products.Product
WHERE   ProductType <> 'snurf'
   OR   (IS_MEMBER('snurfViewer') = 1)
   OR   (IS_MEMBER('db_owner') = 1) --can't add db_owner to a role
WITH CHECK OPTION;
```

Though you probably don't want to give too broad of a permissions to the object (no need to present all users with the possibility of using the view when they can't), you can still give rights to public, and only users mentioned in the code will be able to see data returned. We will simply grant rights to the procedure to public so any authenticated user in the database could use the view.

```
GRANT SELECT ON Products.ProductsSelective TO public;
```

Next, you create a principal named chrissy and the snurfViewer role. Note that you don't add this user to the group yet; you'll do that later in the example:

```
CREATE USER chrissy WITHOUT LOGIN;
CREATE ROLE snurfViewer;
```

Then, you change security context to chrissy and select from the view:

```
EXECUTE AS USER = 'chrissy';
SELECT * FROM Products.ProductsSelective;
REVERT;
```

This returns the one row to which she has access:

```
ProductId   ProductCode   Description          UnitPrice   ActualCost   ProductType
---------   -----------   -----------------    ---------   ----------   -----------
1           widget12      widget number 12     10.5000     8.5000       widget
```

Next, you add Chrissy to the snurfViewer group, go back to context as this user, and run the statement again:

```
ALTER ROLE snurfViewer ADD MEMBER chrissy;
GO

EXECUTE AS USER = 'chrissy';
SELECT *
FROM Products.ProductsSelective;

REVERT;
```

Now, you see all the rows:

```
ProductId   ProductCode   Description          UnitPrice   ActualCost   ProductType
---------   -----------   -----------          ---------   ----------   -----------
1           widget12      widget number 12     10.5000     8.5000       widget
2           snurf98       Snurfulator          99.9900     2.5000       snurf
```

This is even better, but it's still rigid and requires foreknowledge of the data during the design phase. Instead, you'll create a table that maps database role principals with different types of products. Now, this gives you total control. To start the process, we will create the following table:

www.it-ebooks.info

```
CREATE TABLE Products.ProductSecurity
(
  ProductsSecurityId int NOT NULL IDENTITY(1,1)
                CONSTRAINT PKProducts_ProductsSecurity PRIMARY KEY,
  ProductType varchar(20) NOT NULL, --at this point you probably will create a ProductType
                                    --domain table, but this keeps the example a bit simpler
  DatabaseRole sysname NOT NULL,
                CONSTRAINT AKProducts_ProductsSecurity_typeRoleMapping
                        UNIQUE (ProductType, DatabaseRole)
);
```

Then, you insert a row that will be used to give everyone with database rights the ability to see widget-type products:

```
INSERT INTO Products.ProductSecurity(ProductType, DatabaseRole)
VALUES ('widget','public');
```

Next, we alter the `ProductsSelective` view to show only rows to which the user has rights, based on row security:

```
ALTER VIEW Products.ProductsSelective
AS
SELECT  Product.ProductId, Product.ProductCode, Product.Description,
        Product.UnitPrice, Product.ActualCost, Product.ProductType
FROM    Products.Product AS Product
          JOIN Products.ProductSecurity AS ProductSecurity
            ON (Product.ProductType = ProductSecurity.ProductType
                AND IS_MEMBER(ProductSecurity.DatabaseRole) = 1)
                OR IS_MEMBER('db_owner') = 1; --don't leave out the dbo!
```

This view joins the `Product` table to the `ProductSecurity` table and checks the matching roles against the role membership of the principal. Now, you test it:

```
EXECUTE AS USER = 'chrissy';

SELECT *
FROM   Products.ProductsSelective;

REVERT;
```

This returns the following result:

```
ProductId   ProductCode   Description          UnitPrice   ActualCost   ProductType
---------   -----------   ----------------     ---------   ----------   -----------
1           widget12      widget number 12     10.5000     8.5000       widget
```

Then, you add the `snurfViewer` role to the `ProductSecurity` table and try again:

```
INSERT INTO Products.ProductSecurity(ProductType, databaseRole)
VALUES ('snurf','snurfViewer');
GO
EXECUTE AS USER = 'chrissy';
SELECT * FROM Products.ProductSecurity;
REVERT;
```


www.it-ebooks.info

Now, you see it returns all data:

```
ProductId   ProductCode   Description          UnitPrice   ActualCost   ProductType
---------   -----------   ----------------     ---------   ----------   -----------
1           widget12      widget number 12     10.5000     8.5000       widget
2           snurf98       Snurfulator          99.9900     2.5000       snurf
```

This causes a bit of overhead, but then again, all solutions for row-level security will. If you need it, you need it, and not much more can be said. The important aspect of this solution is that we can now use this view in a stored procedure, and regardless of who owns the stored procedure, we can restrict row usage in a generic manner that uses only SQL Server security.

■ **Tip** You can take this type of thing to another level and get really specific with the security. You can even selectively hide and show columns to the user (replacing values with `NULL`s, or `securevalue`, or something using a `CASE` expression like `CASE WHEN IS_MEMBER(..) THEN Column ELSE 'SECURE VALUE' END`). What's better is that once you have set it up, it's a no-brainer to add a principal to a group, and bam, that person has everything he or she needs, without costly setup.

Crossing Database Lines

So far, all the code and issues we've discussed have been concerned with everything owned by a single owner in a single database. When our code and/or relationships must go outside the database limits, the complexity is greatly increased. This is because in SQL Server architecture, databases are generally designed to be thought of as independent containers of data, and this is becoming more and more the expectation with Azure, contained databases, and even the developer tools that are shipped with SQL Server. One database equals one project in so many cases. However, often, you need to share data from one database to another, often for some object that's located in a third-party system your company has purchased.

This can be a real annoyance for the following reasons:

- Foreign key constraints cannot be used to handle referential integrity needs (I covered in Chapter 7 how you implement relationships using triggers to support this).
- Backups must be coordinated or your data could be out of sync with a restore. You lose some of the protection from a single database-backup scenario. This is because when, heaven forbid, a database restore is needed, it isn't possible to make certain that the data in the two databases is in sync.

Although accessing data in outside databases is not optimal, sometimes it's unavoidable. A typical example might be linking an off-the-shelf system into a homegrown system. The off-the shelf package may have requirements that you not make any changes or additions to its database schema. So you create a database to bolt on functionality. Beyond the coding and maintenance aspects, which aren't necessarily trivial, a very important consideration is security. As mentioned in the first paragraph of this section, databases are generally considered independent in the security theme of how SQL Server works. This causes issues when you need to include data outside the database, because users are scoped to a database. That's why `userA` in `database1` is never the same as `userA` in `database2`, even in an uncontained database mapped to the same login.

The ownership chain inside the boundaries of a database is relatively simple. If the owner of the object refers only to other objects owned by that user, then the chain isn't broken. Any user to whom the object's owner grants rights can use the object. However, when leaving the confines of a single database, things get murky. Even

www.it-ebooks.info

if a database is owned by the same system login, the ownership chain is (by default) broken when an object references data outside the database. So, not only does the object creator need to have access to the objects outside the database, the caller needs rights also.

In this section, I will be demonstrating four different concepts when dealing with accessing data outside of a single database:

- Using cross-database chaining
- Using impersonation to implement cross database connections
- Certificate-based trusts
- Accessing data outside of the server

Ideally, you will seldom, if ever, need to access data that is not within the boundaries of a single database, but in practice, that can be an impossible dream. A major example that is common is a third-party application database, where (due to support rules) you are not allowed to put objects in the database, but you need to add to the functionality.

Using Cross-Database Chaining

The cross-database chaining solution is to tell the database to recognize that indeed the owners of `database1` and `database2` are the same. Then, if you, as system administrator, want to allow users to use your objects seamlessly across databases, then it's fine. However, a few steps and requirements need to be met:

- Each database that participates in the chaining relationship must be owned by the same -system login.
- The `DB_CHAINING` database option (set using `ALTER DATABASE`) must be set to `ON` for each database involved in the relationship. It's `OFF` by default.
- The database where the object uses external resources must have the `TRUSTWORTHY` database option set to `ON`; it's `OFF` by default. (Again, set this using `ALTER DATABASE`.)
- The users who use the objects need to have a user in the database where the external resources reside.

I often will use the database chaining approach to support a reporting solution. For example, we have several databases that make up a complete reporting solution in our production system. We have a single database with views of each system to provide a single database for reporting from the OLTP databases (for real-time reporting needs only; other reporting comes from an integrated copy of the database and a data warehouse). Users have access to the views in the reporting database, but not rights to the base database tables.

■ **Caution** If I could put this caution in a flashing font, I would, but my editor would probably say it wasn't cost effective or something silly like that. It's important to understand the implications of the database chaining scenario. You're effectively opening up the external database resources completely to the users in the database who are members of the `db_owner` database role, even if they have no rights in the external database. Because of the last two criteria in the bulleted list, chaining isn't necessarily a bad thing to do for most corporate situations where you simply have to retrieve data from another database. However, opening access to the external database resources can be especially bad for shared database systems, because this can be used to get access to the data in a chaining-enabled database. All that may need to be known is the username and login name of a user in the other database.

---

www.it-ebooks.info

Note that if you need to turn chaining on or off for all databases, you can use `sp_configure` to set `'Cross DB Ownership Chaining'to '1'`, but this is not considered a best practice. Use `ALTER DATABASE` to set chaining *only* where absolutely required.

As an example, consider the following scenario where I'll create two databases with a table in each database and then a procedure. First, I'll create the new database and add a simple table. I won't add any rows or keys, because this isn't important to this demonstration. Note that you have to create a login for this demonstration, because the user must be based on the same login in both databases. We will start with this database in an uncontained model and will switch to a contained model to see how it affects the cross database access:

```
CREATE DATABASE externalDb;
GO
USE externalDb;
GO
                                    --smurf theme song :)
CREATE LOGIN smurf WITH PASSWORD = 'La la, la la la la, la, la la la la';
CREATE USER smurf FROM LOGIN smurf;
CREATE TABLE dbo.table1 ( value int );
```

Next, you create a local database, the one where you'll be executing your queries. You add the login you created as a new user and again create a table:

```
CREATE DATABASE localDb;
GO
USE localDb;
GO
CREATE USER smurf FROM LOGIN smurf;
```

Another step that's generally preferred is to have all databases owned by the same `server_principal`, usually the sysadmin's account. I will use the `sa` account here. Having the databases owned by `sa` prevents issues if the Windows Authentication account is deleted or disabled. We do this with the `ALTER AUTHROIZATION` DDL statement:

```
ALTER AUTHORIZATION ON DATABASE::externalDb TO sa;
ALTER AUTHORIZATION ON DATABASE::localDb TO sa;
```

To check the owner of the database, use the `sys.databases` catalog view:

```
SELECT  name,suser_sname(owner_sid) AS owner
FROM    sys.databases
WHERE   name in ('externalDb','localDb');
```

This should return the following (if not, you will want to make sure of what you may have missed, as it is essential that the databases are owned by the same system principal for the upcoming examples):

```
name                owner
----------          -----
externalDb          sa
localDb             sa
```

Next, you create a simple procedure, still in the `localDb` context, selecting data from the external database, with the objects being owned by the same dbo owner. You then give rights to our new user:

```
CREATE PROCEDURE dbo.externalDb$testCrossDatabase
AS
SELECT value
```


www.it-ebooks.info

```
FROM   externalDb.dbo.table1;
GO
GRANT EXECUTE ON dbo.externalDb$testCrossDatabase TO smurf;
```

Now, try it as the sysadmin user:

```
EXECUTE dbo.externalDb$testCrossDatabase;
```

And it works fine, because the sysadmin user is basically implemented to ignore all security. Execute as the user `smurf` that is in the `localDb`:

```
EXECUTE AS USER = 'smurf';
GO
EXECUTE dbo.externalDb$testCrossDatabase;
GO
REVERT;
```

This will give you the following error:

```
Msg 916, Level 14, State 1, Procedure externalDb$testCrossDatabase, Line 3
The server principal "smurf" is not able to access the database "externalDb" under the current
security context.
```

You then set the chaining and trustworthy attributes for the `localDb` and chaining for the `externalDb` (making these settings requires sysadmin rights):

```
ALTER DATABASE localDb
  SET DB_CHAINING ON;
ALTER DATABASE localDb
  SET TRUSTWORTHY ON;
ALTER DATABASE externalDb
  SET DB_CHAINING ON;
```

Now, if you execute the procedure, you will see that it returns a valid result. This is because

- The owner of the objects and databases is the same, which we set up with the `ALTER AUTHORIZATION` statements.
- The user has access to connect to the external database, which was why we created the user when we set up the `externalDb` database. (You can also use the guest login to allow any user to access the database as well, though as mentioned, this is not a best practice).

You can validate the metadata for these databases using the `sys.databases` catalog view:

```
SELECT  name, is_trustworthy_on, is_db_chaining_on
FROM    sys.databases
WHERE   name in ('externalDb','localDb');
```

This returns the following results:

```
name            is_trustworthy_on       is_db_chaining_on
----------      -----------------       -----------------
externalDb      0                       1
localDb         1                       1
```


www.it-ebooks.info

I find that the biggest issue when setting up cross-database chaining is the question of ownership of the databases involved. The owner changes sometimes because users create databases and leave them owned by their security principals. Note that this is the only method I will demonstrate that doesn't require stored procedures to work. You can also use basic queries and views in this very same manner, as they are simply stored queries that you use as the basis of a SELECT statement. Stored procedures are executable code modules that allow them a few additional properties, which I will demonstrate in the next two sections.

Now let's see how this will be affected by setting the database to use the containment model:

```
ALTER DATABASE localDB SET CONTAINMENT = PARTIAL;
```

Then, connect to the server as user: smurf using SSMS and default database to localDb and try to run the procedure. You will notice that the connection isn't to the contained database it is to the server and you are in the context of the contained database. Using EXECUTE AS will give you the same effect:

```
EXECUTE AS USER = 'smurf';
go
EXECUTE dbo.externalDb$testCrossDatabase;
GO
REVERT;
GO
```

You will see that it behaves the exact same way and gives you a result to the query. However, connect with a contained user is a different challenge. First, let's create a contained user, and then give it rights to execute the procedure.

```
CREATE USER Gargy WITH PASSWORD = 'Nasty1$';
GO
GRANT EXECUTE ON dbo.externalDb$testCrossDatabase TO Gargy;
```

Next, change to the database security context of the new contained user and try to change context to the externalDb:

```
EXECUTE AS USER = 'Gargy';
GO
USE externalDb;
```

This will give the following error:

```
Msg 916, Level 14, State 1, Line 1
The server principal "S-1-9-3-3326261859-1215110459-3885819776-190383717." is not able to
access the database "externalDb" under the current security context.
```

Obviously the "server principal" part of the error message could be confusing, but it is also true because in this case, the database will behave as a server to that user. Executing the following code will give you the exact same error:

```
EXECUTE dbo.externalDb$testCrossDatabase;
GO
REVERT;
GO
```

When turning on containment, you will note that since the maximum containment level is PARTIAL, some code you have written may not be containment safe. To check, you can use the sys.dm_db_uncontained_entities

www.it-ebooks.info

dynamic management view. Even if you aren't using containment, you can use the following containment DMVs query to see code referencing outside data:

```
SELECT  OBJECT_NAME(major_id) AS object_name,statement_line_number,
        statement_type, feature_name, feature_type_name
FROM    sys.dm_db_uncontained_entities AS e
WHERE   class_desc = 'OBJECT_OR_COLUMN';
```

For our database, it will return the following, which corresponds to the procedure we created and the query that used a cross-database reference:

```
object_name                      statement_line_number      statement_type
-------------------------        ---------------------      --------------
externalDb$testCrossDatabase     3                          SELECT

feature_name                          feature_type_name
-------------------------------       -----------------
Server or Database Qualified Name     T-SQL Syntax
```

The object will also return uncontained users:

```
SELECT  USER_NAME(major_id) AS USER_NAME,*
FROM    sys.dm_db_uncontained_entities AS e
WHERE   class_desc = 'DATABASE_PRINCIPAL'
  and   USER_NAME(major_id) <> 'dbo';
```

And we created one already in this chapter:

```
USER_NAME
---------
smurf
```

■ **Note** One additional very interesting (albeit non-security-related) feature of contained databases is that the collation of the `tempdb` as seen from the contained user will be that of the contained database. While this is not frequently an issue for most databases, it will make life easier for moving databases around to servers with different collations. I won't cover that feature in any other location in this book.

Finally, we need to do a bit of housekeeping to remove containment from the database. To do this, we will delete the contained user we created and turn off containment (we have to drop the user, or as previously mentioned, you would receive an error stating that uncontained databases cannot have contained users):

```
DROP USER Gargy;
GO
USE Master;
GO
ALTER DATABASE localDB SET CONTAINMENT = NONE;
```


www.it-ebooks.info

```
GO
USE LocalDb;
GO
```

Using Impersonation to Cross Database Lines

Impersonation can be an alternative to using the DB_CHAINING setting. Now, you no longer need to set the chaining to ON; all you need is to set it to TRUSTWORTHY:

```
ALTER DATABASE localDb
  SET DB_CHAINING OFF;
ALTER DATABASE localDb
  SET TRUSTWORTHY ON;

ALTER DATABASE externalDb
 SET DB_CHAINING OFF;
```

Now, you can rewrite the procedure like this, which lets the person execute in the context of the owner of the schema that the procedure is in:

```
CREATE PROCEDURE dbo.externalDb$testCrossDatabase_Impersonation
WITH EXECUTE AS SELF --as procedure creator, who is the same as the db owner
AS
SELECT  value
FROM    externalDb.dbo.table1;
GO
GRANT execute on dbo.externalDb$testCrossDatabase_impersonation TO smurf;
```

If the login of the owner of the dbo schema (in this sa, because I set the owner of both databases to sa) has access to the other database, you can impersonate dbo in this manner. In fact, you can access the external resources seamlessly. This is probably the simplest method of handling cross-database chaining for most corporate needs. Of course, impersonation should be used very carefully and raise a humongous flag if you're working on a database server that's shared among many different companies.

Now, when I execute the procedure as smurf user, it works:

```
EXECUTE AS USER = 'smurf';
GO
EXECUTE dbo.externalDb$testCrossDatabase_impersonation;
GO
REVERT;
```

If you toggle off TRUSTWORTHY and try to execute the procedure

```
ALTER DATABASE localDb SET TRUSTWORTHY OFF;
GO
EXECUTE dbo.externalDb$testCrossDatabase_impersonation;
```

no matter what user you execute as, you'll receive the following error:

```
Msg 916, Level 14, State 1, Procedure externalDb$testCrossDatabase_Impersonation, Line 4
The server principal "sa" is not able to access the database "externalDb" under the current
security context.
```


www.it-ebooks.info

This is clearly another of the confusing sorts of error messages you get on occasion, since the server principal sa ought to be able to do anything, but it is what it is. Next, we go back to the containment method. Turn back on TRUSTWORTHY, set the containment, and re-create the Gargy user, giving rights to the impersonation procedure:

```
ALTER DATABASE localDb SET TRUSTWORTHY ON;
GO
ALTER DATABASE localDB SET CONTAINMENT = PARTIAL;
GO
CREATE USER Gargy WITH PASSWORD = 'Nasty1$';
GO
GRANT EXECUTE ON externalDb$testCrossDatabase_Impersonation TO Gargy;
```

Now execute the procedure in the context of the contained user:

```
EXECUTE AS USER = 'Gargy';
GO
EXECUTE dbo.externalDb$testCrossDatabase_Impersonation;
GO
REVERT;
```

This time, you will see that there no error is raised, because the procedure is in the context of the owner of the procedure and is mapped to a server principal that is the same that owns the database and the object you are using. Note that this breaks (or really, violates) containment because you are using external data, but it will give you the rights you need in the (hopefully) rare requirement to use cross-database access.

Finally, we clean up the users and containment as we have done before:

```
DROP USER Gargy;
GO
USE Master;
GO
ALTER DATABASE localDB SET CONTAINMENT = NONE;
GO
USE LocalDb;
```

Using a Certificate-Based Trust

The final thing I'll demonstrate around cross-database access is using a single certificate installed in both databases to let the code access data across database boundaries. We'll use it to sign the stored procedure and map a user to this certificate in the target database. This is a straightforward technique and is the best way to do cross-database security chaining when the system isn't a dedicated corporate resource. It takes a bit of setup, but it isn't overwhelmingly difficult. What makes using a certificate nice is that you don't need to open the hole left in the system's security by setting the database to TRUSTWORTHY. This is because the user who will be executing the procedure is a user in the database, just as if the target login or user were given rights in the externalDB. Because the certificate matches, SQL Server knows that this cross-database access is acceptable.

First, turn off the TRUSTWORTHY setting:

```
REVERT;
GO
USE localDb;
GO
```


www.it-ebooks.info

```
ALTER DATABASE localDb
    SET TRUSTWORTHY OFF;
```

Check the status of your databases as follows:

```
SELECT name,
       SUSER_SNAME(owner_sid) as owner,
       is_trustworthy_on, is_db_chaining_on
FROM   sys.databases where name IN ('localdb','externaldb');
```

This should return the following results (if not, go back and turn off TRUSTWORTHY and chaining for the databases where necessary):

```
name           owner     is_trustworthy_on     is_db_chaining_on
----------     -----     -----------------     -----------------
externalDb     sa        0                     0
localDb        sa        0                     0
```

Now, we will create another procedure and give the user smurf rights to execute it, just like the others (which won't work now because TRUSTWORTHY is turned off):

```
CREATE PROCEDURE dbo.externalDb$testCrossDatabase_Certificate
AS
SELECT  Value
FROM    externalDb.dbo.table1;
GO
GRANT EXECUTE on dbo.externalDb$testCrossDatabase_Certificate to smurf;
```

Then, create a certificate:

```
CREATE CERTIFICATE procedureExecution ENCRYPTION BY PASSWORD = 'jsaflajOIo9jcCMd;SdpSljc'
  WITH SUBJECT =
        'Used to sign procedure:externalDb$testCrossDatabase_Certificate';
```

Add this certificate as a signature on the procedure:

```
ADD SIGNATURE TO dbo.externalDb$testCrossDatabase_Certificate
  BY CERTIFICATE procedureExecution WITH PASSWORD = 'jsaflajOIo9jcCMd;SdpSljc';
```

Finally, make an OS file out of the certificate, so a certificate object can be created in the externalDb based on the same certificate (choose a directory that works best for you):

```
BACKUP CERTIFICATE procedureExecution TO FILE = 'c:\temp\procedureExecution.cer';
```

This completes the setup of the localDb. Next, you have to apply the certificate to the externalDb:

```
USE externalDb;
GO
CREATE CERTIFICATE procedureExecution FROM FILE = 'c:\temp\procedureExecution.cer';
```

After that, map the certificate to a user, and give this user rights to the table1 that the user in the other database is trying to access:

```
CREATE USER procCertificate FOR CERTIFICATE procedureExecution;
GO
GRANT SELECT on dbo.table1 TO procCertificate;
```


www.it-ebooks.info

Now, you're good to go. Change back to the `localDb` and execute the procedure:

```
USE localDb;
GO
EXECUTE AS LOGIN = 'smurf';
EXECUTE dbo.externalDb$testCrossDatabase_Certificate;
```

The stored procedure has a signature that identifies it with the certificate, and in the external database, it connects with this certificate to get the rights of the certificate-based user. So, since the certificate user can view data in the table, your procedure can use the data.

The certificate-based approach isn't as simple as the other possibilities, but it's more secure, for certain. Pretty much the major downside to this is that it works only with procedures and not with views, and I didn't really cover any of the intricacies of using certificates. However, now you have a safe way of crossing database boundaries that doesn't require giving the user direct object access and doesn't open up a hole in your security. Hence, you could use this solution on any server in any situation. Make sure to secure or destroy the certificate file once you've used it, so no other user can use it to gain access to your system. Then you clean up the databases used for the example.

Of the methods shown, this would be the least desirable to use with containment, because you now have even more baggage to set up after moving the database, so we will simply leave it as "it could be done, but shouldn't be."

Finally, we clean up the databases used for the examples and move back to the `ClassicSecurityExample` database we have used throughout this chapter:

```
REVERT;
GO
USE MASTER;
GO
DROP DATABASE externalDb;
DROP DATABASE localDb;
GO
USE ClassicSecurityExample;
```

Different Server (Distributed Queries)

I want to make brief mention of distributed queries and introduce the functions that can be used to establish a relationship between two SQL Server instances, or a SQL Server instance and an OLE DB or ODBC data source. (Note that OLEDB is being deprecated and will not be supported for long. For more details check `http://msdn.microsoft.com/en-us/library/ms810810.aspx`, which outlines Microsoft's "Data Access Technologies Road Map.")

You can use either of these two methods:

- *Linked servers*: You can build a connection between two servers by registering a "server" name that you then access via a four-part name (`<linkedServerName>.<database>.<schema>.<objectName>`) or through the `OPENQUERY` interface. The linked server name is the name you specify using `sp_addlinkedserver`. This could be a SQL Server server or anything that can be connected to via OLE DB.
- *Ad hoc connections*: Using the `OPENROWSET` or `OPENDATASOURCE` interfaces, you can return a table of data from any OLE DB source.

In either case, the security chain will be broken when crossing SQL Server instance connections and certainly when using any data source that isn't SQL Server–based. Whether or not this is a problem is based on the connection properties and/or connection string used to create the linked server or ad hoc connection. Using

www.it-ebooks.info

linked servers, you could be in the context of the Windows login you are logged in with, a SQL Server standard login on the target machine, or even a single login that everyone uses to "cross over" to the other server. The best practice is to use the Windows login where possible.

As I mentioned briefly in the previous section, one use for `EXECUTE AS` could be to deal with the case where you're working with distributed databases. One user might be delegated to have rights to access the distributed server, and then, you execute the procedure as this user to give access to the linked server objects.

Using linked servers or ad-hoc connections will both break the containment model. Linked servers are defined in the master database.

Obfuscating Data

It isn't always possible to keep users from accessing data. We database administrator types all too often have unfettered access to entire production systems with far too much personal data. Even with the far-improved security granularity starting with SQL Server 2005, a few users will still have rights to run as a member of the `sys admin` server role, giving them access to *all* data.

Also, technically, once people access to unencrypted backups of the system, they can easily access any data in the database by simply restoring it to a different server where they are administrators. And don't forget that if you can attach the database, the data is also going to be viewable, in some fashion. If you're dealing with sensitive data, you need to be wary of how you deal with this data:

- Do you back up the database? Where are these backups?
- Do you send the backups to an offsite safe location? Who takes them there?
- Who has access to the servers where data is stored? Do you trust the fate of your company in their hands? Could these servers be hacked?

SQL Server definitely improved on the 2008 version in terms of the DBA job of protecting data by implementing *transparent data encryption*. Basically, this will encrypt the data and log files during I/O so that you don't have to change your code, but anytime the data is at rest, it will stay encrypted (including when it is backed up). You can also use several third-party tools to encrypt backups.

When data is at rest in the database and users have access to the data, it is also important that we obfuscate the data such that a user cannot tell what it is exactly. This is one of the main ways that we can protect data from casual observers, especially those like us DBA types who generally have full control over the database (in other words, way too much power in the database).

The biggest "key" to encryption is that you don't include all the information needed to decrypt the data easily available to the user. SQL Server has a rich set of encryption features, featuring a key and certificate management system. You can use this system to encrypt data from each user and from people who might get access to the data outside of the server.

In the following example, I'll use the simplest encryption that SQL Server gives us, using the `encryptByPassPhrase`. (You can also use keys, asymmetric keys, and certificates.) The `encryptByPassPhrase` function lets you specify your key as a string value that SQL Server uses to munge the data, such that you must have this key to reconstitute the data. Passphrase encryption can be one of the most useful forms of encryption using the SQL Server built-in encryption, because you can let each row have its own password. So, let's say we have a column of very personal information, such as a vault of information that an end user might be able to save securely. Using the password in the application, the data could become visible, but it would not be available to anyone else who did not know the password. Of course, passphrase encryption does suffer from the problem that if the password is lost, the data is lost for good.

As an example, we'll build an incredibly simple encryption scheme that you could use to secure a column in your database. First, as a quick demonstration of how the function works, consider the following:

```
SELECT encryptByPassPhrase('hi', 'Secure data');
```


www.it-ebooks.info

'Secure data' is the value to be encrypted, and 'hi' is the passphrase that's required to get this value back later. Executing this statement returns the following result (the results will be different each time you execute the statement, but if you take my results or yours, you will get back the answer 'Secure data'):

```
------------------------------------------------------------------------
0x010000004D2B87C6725612388F8BA4DA082495E8C836FF76F32BCB642B36476594B4F014
```

This is a clearly unreadable binary string, and even cooler, the value is different every time you execute it, so no one can come behind and decrypt it with any ease. To decrypt it, just use the following:

```
SELECT decryptByPassPhrase('hi',
0x010000004D2B87C6725612388F8BA4DA082495E8C836FF76F32BCB642B36476594B4F014);
```

This returns the following result:

```
------------------------
0x5365637572652064617461
```

This is the binary representation of our original string, which makes it easy to represent most any datatype. To use the value, you have to cast it back to the original varchar type:

```
SELECT CAST(decryptByPassPhrase('hi',
  0x010000004D2B87C6725612388F8BA4DA082495E8C836FF76F32BCB642B36476594B4F014)
    AS VARCHAR(30));
```

This returns the following result:

```
-----------
Secure data
```

■ **Tip** The data is different each time because the encryption scheme uses various nonencrypted data, such as the time value, to encrypt the data (this is generally known as the *salt value* of the encryption). You cannot infer the decryption of all values based on breaking a single value. This is also what makes searching encrypted data so difficult (as it really should be!).

I am not going to go any deeper into the concepts of encryption, because it is far too complex to cover in one section of a chapter (a couple of my great guys, Micheal Coles and Rodney Landrum have an Apress book on the subject named *Expert SQL Server 2008 Encryption* that will give you an understanding of why it is an entire book worth of material). The important thing to do for your database design is to make sure that any time you store personally identifiable data or really any data that could harm the customer if it were to get out should be encrypted. Social Security numbers are a prime example. If your database needs to store such data it should be stored encrypted.

Keep in mind as you design your encryption needs, "Who can decrypt the data?" If the information to do the decryption is stored with the encrypted data, you are not gaining much. SQL Server's basic encryption uses

www.it-ebooks.info

certificates and keys that are all on the local server. This can keep employees eyes off of sensitive data, but if a person got the entire database and a sysadmin role user, they could get to everything. SQL Server has a better method called extensible key management, but it still isn't perfect. The best method is to implement your decryption engine on a different server that is not linked to the database engine at all.

A very useful and security-minded goal you should have in most organizations is to try not to hold information like credit card numbers in your database at all and use a third-party bank to handle it for you. Of course, if you do work for a bank, I clearly could not do your encryption needs justice in a single chapter, much less this section.

In the end, if your backup tape is pocketed by an employee who's changing jobs from legit IT worker to a personal identification salesperson, the data is worthless, and all your press release need say is "Company X regrets to say that some financial data tapes were stolen from our vaults. All data was encrypted; hence, there is little chance any usable data was acquired." And you, as DBA, will be worshipped—and isn't that really the goal of every programmer!

Monitoring and Auditing

Often, a client won't care too much about security, so he or she doesn't want to limit what a user can do in the database. However, many times, there's a hidden subtext: "I don't want to be restrictive, but how can we keep up with what users have done?"

An alternative to implementing a full-blown security system can be simply to watch what users do, in case they do something they shouldn't. To implement our Big Brother security scenario, I'll demonstrate three possible techniques:

- *Server and database audit:* In SQL Server 2008 and later, you can define audit specifications that will watch and log all the activity as you define.
- *Watching table changes using DML triggers*: Keep up with a history of previous values for rows in a table.
- *DDL triggers*: Use these to log any DDL events, such as creating users, creating or modifying tables, and so on.
- *Using Profiler*: Profiler catches everything, so you can use it to watch user activity.

In some cases, these techniques are used to augment the security of a system, for example, to make sure system administrators keep out of certain parts of the system.

■ **Note** In SQL Server 2008, Microsoft introduced two other new features that are interesting for watching changes to the database, but they are not of direct value for a security purpose. Change Data Capture is an Enterprise Edition–only feature that allows you to do full tracking of every change in data, and change tracking is available to other editions to capture that a change has occurred since the last time you checked the Change Tracking system. However, neither of those new features will tell you the user who made a given change; thus, they don't have security applications. They are, however, amazingly useful for implementing a reporting/data warehouse system, because finding the rows that changed for ETL has always been the hardest thing. These new features have less overhead (and certainly less setup) than do triggers that you might write to accomplish the same goal.

---

www.it-ebooks.info

Server and Database Audit

For the 2008 release, Microsoft added a very nice feature for auditing the activities of your users. It is called SQL Server Audit, and it allows you to define server- and database-level audits that you can use to monitor almost everything your logins and users do to your server and/or databases. For SQL Server 2008, it was a feature that is included only in the Enterprise Edition. For SQL Server 2012, database-level audits are still Enterprise only, but server-level audits are available in all versions to monitor server configuration changes.

In versions before SQL Server 2008, most monitoring would have required a web of DML and DDL triggers, plus the use of Profiler to watch everyone's moves. Now, you can implement detailed monitoring in a declarative manner. SQL Server Audit is a tremendously cool feature and will make the process of meeting auditing requirements *much* easier than ever before. Instead of lots of code to pore through, you can just print the audits that you are enforcing, and you are done. Note that SQL Server Audit does not obviate the use of DML or DDL triggers as a tool for watching what users are doing, mostly because in a trigger you can react to what a user is doing and alter their path. Using SQL Server Audit, you will simply be able to watch what the user is doing.

■ **Note** As usual, there are graphical user interface (GUI) versions of everything I discuss, and I imagine that many DBAs (even some hardcore ones) will probably use the GUI for the most part, but as with everything else in this book, I want to show the syntax because it will make using the GUI easier, and if you have to apply these settings to more than one server, you will quickly learn to write scripts, or at least to use the GUI to generate scripts.

The auditing is file based, in that you don't do your logging to a database; rather, you specify a directory on your server (or off your server if you so desire). You will want to make sure that the directory is a very fast access location to write to, because writing to it will be part of the transactions you execute. When auditing is turned on, each operation will be audited or not executed. However, it doesn't write directly to the file; rather, for maximum performance, SQL Server Audit uses Service Broker queues under the covers, so it doesn't have to write audit data to the file as part of each transaction. Instead, queue mechanisms make sure that the data is written asynchronously (there is a setting to force an audit trail to be written in some amount of time or synchronously if you need it to be guaranteed 100 percent up to date).

The audit structures consist of three basic objects:

- *Server audit*: Top-level object that defines where the audit file will be written to and other essential settings
- *Server audit specification*: Defines the actions at the server level that will be audited
- *Database audit specification*: Defines the actions at the database level that will be audited

In the following sections, we will go through the steps to define an audit specification, enable the audit, and then view the audit results.

Defining an Audit Specification

As an example, I will set up an audit on our test server/security database to watch for logins to be changed (such as a new login created or one changed/dropped) as well as watching for the employee or manager user to execute a `SELECT` statement against the `Products.Product` table and `SELECT`s by anyone on `Sales.Invoice`. First, you define the `SERVER AUDIT`:

```
USE master;
GO
```


www.it-ebooks.info

```
CREATE SERVER AUDIT ProSQLServerDatabaseDesign_Audit
TO FILE                          --choose your own directory, I expect most people
(       FILEPATH = N'c:\temp\' --have a temp directory on their system drive
        ,MAXSIZE = 15 MB
        ,MAX_ROLLOVER_FILES = 0 --unlimited
)
WITH
(
        ON_FAILURE = SHUTDOWN --if the file cannot be written to,
                              --shut down the server
);
```

■ **Note** The audit is created in a disabled state. You need to start it once you have added audit specifications.

The next step is to define an audit specification to set up the container to hold a list of related items to audit. This container-based approach lets you easily enable or disable auditing for the entire group of related features. Create the container by defining a `SERVER AUDIT SPECIFICATION`:

```
CREATE SERVER AUDIT SPECIFICATION ProSQLServerDatabaseDesign_Server_Audit
        FOR SERVER AUDIT ProSQLServerDatabaseDesign_Audit
        WITH (STATE = OFF); --disabled. I will enable it later
```

The next step is to add things to the specification to audit. There are lots of different things you can audit. You can find the list under "SQL Server Audit Action Groups and Actions" in Books Online. In our sample, we are going to watch for server principals to change:

```
ALTER SERVER AUDIT SPECIFICATION ProSQLServerDatabaseDesign_Server_Audit
  ADD (SERVER_PRINCIPAL_CHANGE_GROUP);
```

Next, I will go through the same process for the database that I did for the server, setting up the container for the audit using the `DATABASE AUDIT SPECIFICATION` command:

```
USE ClassicSecurityExample;
GO
CREATE DATABASE AUDIT SPECIFICATION
        ProSQLServerDatabaseDesign_Database_Audit
  FOR SERVER AUDIT ProSQLServerDatabaseDesign_Audit
  WITH (STATE = OFF);
```

This time, we will audit the employee and manager database principals use of the `Products.Product` table. Here is how we add those items to the specification:

```
ALTER DATABASE AUDIT SPECIFICATION
  ProSQLServerDatabaseDesign_Database_Audit
  ADD (SELECT ON Products.Product BY employee, manager),
  ADD (SELECT ON Products.AllProducts BY employee, manager);
```


www.it-ebooks.info

Enabling an Audit Specification

Finally, we enable the two audit specifications that we've just created. Remember, to enable a specification is to enable all the audits defined in that container, for example:

```
USE master;
GO
ALTER SERVER AUDIT ProSQLServerDatabaseDesign_Audit
  WITH (STATE = ON);
ALTER SERVER AUDIT SPECIFICATION ProSQLServerDatabaseDesign_Server_Audit
  WITH (STATE = ON);
GO
USE ClassicSecurityExample;
GO
ALTER DATABASE AUDIT SPECIFICATION ProSQLServerDatabaseDesign_Database_Audit
  WITH (STATE = ON);
```

Viewing the Audit Trail

Now that our audits are enabled, we can monitor the usage of the features and functionality that we're auditing. The following code executes some actions that will be audited as a result the specifications we've just created. The following script will do a few actions that will be audited by the audit objects we have set up in the previous sections:

```
CREATE LOGIN MrSmith WITH PASSWORD = 'Not a good password';
GO
EXECUTE AS USER = 'manager';
GO
SELECT *
FROM   Products.Product;
GO
SELECT *
FROM   Products.AllProducts; --Permissions will fail
GO
REVERT
GO
EXECUTE AS USER = 'employee';
GO
SELECT *
FROM   Products.AllProducts; --Permissions will fail
GO
REVERT;
GO
```

The following query will let us view the log that was set up with the `CREATE SERVER AUDIT` command in the first step of the process. By executing this

```
SELECT event_time, succeeded,
       database_principal_name, statement
FROM sys.fn_get_audit_file ('c:\temp\*',DEFAULT,DEFAULT);
```

we can see the different statements that were executed (and you see the two statements where the permission failed, which is the only reason the data will be put in the audit when it doesn't succeed):

www.it-ebooks.info

```
event_time              succeeded   database_principal_name   statement
----------------------  ---------   ----------------------   ------------------------------
2011-09-02 03:36:53.31  1           dbo                      CREATE LOGIN MrSmith WITH PASS…
2011-09-02 03:36:53.37  1           Manager                  SELECT *
                                                             FROM   Products.Product
2011-09-02 03:36:53.58  0           Manager                  SELECT *
                                                             FROM   Products.AllProducts…
2011-09-02 03:36:53.60  0           Employee                 SELECT *
                                                             FROM   Products.AllProducts…
```

There are lots of other pieces of information returned by the `sys.fn_get_audit_file` function that are very useful, especially including the server principal information. Using a few of the catalog views, you can get a picture of what the audits do. Note that the query I built works only at an object level. It could be extended if you wanted to do column-level audits.

Viewing the Audit Configuration

Finally, once you have set up the audit trail, it is often important to find out what is being audited. You can do this using several of the catalog views:

- `sys.server_audits`: One row per server audit
- `sys.server_audit_specifications`: Details about the audits that have been configured for this server, such as when it was started, the last time it was modified, and so on
- `sys.server_audit_specification_details`: Links the objects being audited and actions being audited

The following query, using these views, will get you the definition of what is being audited at a server level:

```
SELECT  sas.name AS audit_specification_name,
        audit_action_name
FROM    sys.server_audits AS sa
          JOIN sys.server_audit_specifications AS sas
            ON sa.audit_guid = sas.audit_guid
          JOIN sys.server_audit_specification_details AS sasd
            ON sas.server_specification_id = sasd.server_specification_id
WHERE   sa.name = 'ProSQLServerDatabaseDesign_Audit';
```

By executing this, given all of the audit stuff we had set up, will return the following:

```
audit_specification_name                    audit_action_name
------------------------------------        ----------------------------
ProSQLServerDatabaseDesign_Server_Audit     SERVER_PRINCIPAL_CHANGE_GROUP
```

Digging deeper, to get the objects and actions, the following query will get you the database-level actions that are being audited:

```
SELECT  --sas.name AS audit_specification_name,
        audit_action_name,dp.name AS [principal],
        SCHEMA_NAME(o.schema_id) + '.' + o.name AS object
```


www.it-ebooks.info

```
FROM    sys.server_audits as sa
          join sys.database_audit_specifications AS sas
            on sa.audit_guid = sas.audit_guid
          join sys.database_audit_specification_details AS sasd
            on sas.database_specification_id = sasd.database_specification_id
          join sys.database_principals AS dp
            on dp.principal_id = sasd.audited_principal_id
          join sys.objects AS o
            on o.object_id = sasd.major_id
WHERE sa.name = 'ProSQLServerDatabaseDesign_Audit'
  and sasd.minor_id = 0; --need another query for column level audits
```

This query returns the following:

```
audit_action_name       principal           object
-----------------       ---------           --------------------
SELECT                  Employee            Products.Product
SELECT                  Manager             Products.Product
SELECT                  Employee            Products.allProducts
SELECT                  Manager             Products.allProducts
```

Quite a few more catalog views pertain to the server and database auditing facilities of SQL Server, certainly more than is necessary in this chapter for me to cover. The basic setup of auditing is really quite straightforward, and auditing is a nice feature of SQL Server that is going to be a welcome tool for users who have the need to audit the activities of their users and especially administrators.

Watching Table History Using DML Triggers

Even in addition to the kind of logging you can do with SQL Server Audit, you will still find uses for trigger-based logging of table history. And in previous versions of SQL Server, trigger-based logging is very much the way to watch what your users do with your data.

As discussed in Chapter 6, you can run code whenever a user executes an `INSERT`, `UPDATE`, or `DELETE` DML statement on a table or view. We already constructed an auditing trigger in Chapter 6 in the section "DML Triggers" that audited change, and we'll create another one here. When finished, a history of the previous values for a column in a table will be maintained, in case some user changes the data improperly.

The scenario is that we have a slice of the `Sales` and `Inventory` sections of the database for products and invoices (see Figure 9-6).

On each invoice line item, there's a cost and a discount percentage. If the cost value doesn't match the current value in the `Product` table and the discount percentage isn't zero, we want to log the difference. A report of differences will be built and sent to the manager to let the values be checked to make sure everything is within reason.

First we build the tables, going back to the `ClassicSecurityExample` database:

```
USE ClassicSecurityExample;
GO
CREATE SCHEMA Sales;
GO
CREATE SCHEMA Inventory;
GO
```


www.it-ebooks.info

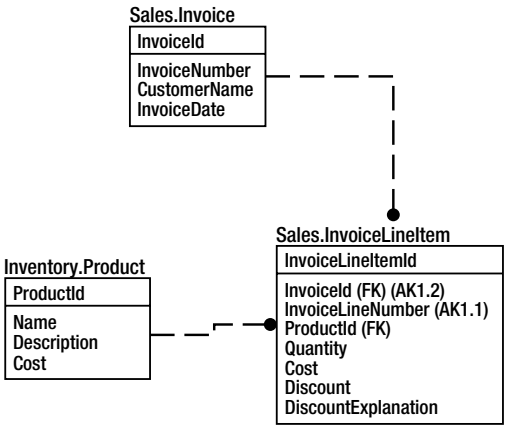


Figure 9-6. *Sample tables for table history example*

```
CREATE TABLE Sales.Invoice
(
  InvoiceId int NOT NULL IDENTITY(1,1) CONSTRAINT PKInvoice PRIMARY KEY,
  InvoiceNumber char(10) NOT NULL
        CONSTRAINT AKInvoice_InvoiceNumber UNIQUE,
  CustomerName varchar(60) NOT NULL , --should be normalized in real database
  InvoiceDate smalldatetime NOT NULL
);
CREATE TABLE Inventory.Product
(
  ProductId int NOT NULL IDENTITY(1,1) CONSTRAINT PKProduct PRIMARY KEY,
  name varchar(30) NOT NULL CONSTRAINT AKProduct_name UNIQUE,
  Description varchar(60) NOT NULL ,
  Cost numeric(12,4) NOT NULL
);
CREATE TABLE Sales.InvoiceLineItem
(
  InvoiceLineItemId int NOT NULL IDENTITY(1,1)
                    CONSTRAINT PKInvoiceLineItem PRIMARY KEY,
  InvoiceId int NOT NULL,
  ProductId int NOT NULL,
  Quantity numeric(6,2) NOT NULL,
  Cost numeric(12,4) NOT NULL,
  discount numeric(3,2) NOT NULL,
  discountExplanation varchar(200) NOT NULL,
  CONSTRAINT AKInvoiceLineItem_InvoiceAndProduct
             UNIQUE (InvoiceId, ProductId),
  CONSTRAINT FKSales_Invoice$listsSoldProductsIn$Sales_InvoiceLineItem
             FOREIGN KEY (InvoiceId) REFERENCES Sales.Invoice(InvoiceId),
  CONSTRAINT FKSales_Product$isSoldVia$Sales_InvoiceLineItem
             FOREIGN KEY (InvoiceId) REFERENCES Sales.Invoice(InvoiceId)
  --more constraints should be in place for full implementation
);
```


www.it-ebooks.info

Now, we create another table to hold the audit of the line item of an invoice:

```
CREATE TABLE Sales.InvoiceLineItemDiscountAudit
(
  InvoiceLineItemDiscountAudit int NOT NULL IDENTITY(1,1)
        CONSTRAINT PKInvoiceLineItemDiscountAudit PRIMARY KEY,
  InvoiceId int NOT NULL,
  InvoiceLineItemId int NOT NULL,
  AuditTime   datetime NOT NULL,
  SetByUserId sysname NOT NULL,
  Quantity numeric(6,2) NOT NULL,
  Cost numeric(12,4) NOT NULL,
  Discount numeric(3,2) NOT NULL,
  DiscountExplanation varchar(300) NOT NULL
);
```

I used a surrogate primary key with no other uniqueness criteria here for the audit table, because you just cannot predict whether two users could change the same row at the same point in time (heck, even the same user could change the row twice at the same time on two different connections!). In this case, we settle on the surrogate to simply represent the order of events, rather than their existing logical uniqueness.

Then, we code a trigger, using the same trigger template as in the previous chapters (and defined in Appendix B). The trigger will cascade the change from the primary table to the audit table behind the scenes:

```
CREATE TRIGGER Sales.InvoiceLineItem$insertAndUpdateAuditTrail
ON Sales.InvoiceLineItem
AFTER INSERT,UPDATE AS
BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
        @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  --    @rowsAffected int = (SELECT COUNT(*) FROM deleted);
  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;
  BEGIN TRY
    --[validation blocks]
    --[modification blocks]
    IF UPDATE(Cost)

    INSERT INTO Sales.InvoiceLineItemDiscountAudit (InvoiceId,
            InvoiceLineItemId, AuditTime, SetByUserId, Quantity,
            Cost, Discount, DiscountExplanation)
    SELECT inserted.InvoiceId, inserted.InvoiceLineItemId,
       current_timestamp, suser_sname(), inserted.Quantity,
       inserted.Cost, inserted.Discount,
       inserted.DiscountExplanation
```


www.it-ebooks.info

```
    FROM inserted
        JOIN Inventory.Product as Product
          ON inserted.ProductId = Product.ProductId
    --if the Discount is more than 0, or the cost supplied is less than the
    --current value
    WHERE inserted.Discount > 0
       OR inserted.Cost < Product.Cost;
          -- if it was the same or greater, that is good!
          -- this keeps us from logging if the cost didn't actually
          -- change

  END TRY
  BEGIN CATCH
        IF @@TRANCOUNT > 0
          ROLLBACK TRANSACTION;
        THROW;
  END CATCH
END
```

We then test the code by creating a few products:

```
INSERT INTO Inventory.Product(name, Description,Cost)
VALUES  ('Duck Picture','Picture on the wall in my hotelRoom',200.00),
        ('Cow Picture','Picture on the other wall in my hotelRoom',150.00);
```

Then, we start an invoice:

```
INSERT INTO Sales.Invoice(InvoiceNumber, CustomerName, InvoiceDate)
VALUES ('IE00000001','The Hotel Picture Company','2012-01-01');
```

Next, we add an `InvoiceLineItem` that's clean, has the same price, and has no discount:

```
INSERT INTO Sales.InvoiceLineItem(InvoiceId, ProductId, Quantity,
                                  Cost, Discount, DiscountExplanation)
SELECT  (SELECT InvoiceId
         FROM   Sales.Invoice
         WHERE  InvoiceNumber = 'IE00000001'),
        (SELECT ProductId
         FROM   Inventory.Product
         WHERE  Name = 'Duck Picture'), 1,200,0,'';
```

We check our log:

```
SELECT * FROM Sales.InvoiceLineItemDiscountAudit;
```

Nothing is returned on insert:

```
InvoiceLineItemDiscountAudit    InvoiceId       InvoiceLineItemId       AuditTime
----------------------------    ---------       -----------------       ---------

SetByUserId     Quantity        Cost    Discount        DiscountExplanation
-----------     --------        ----    --------        -------------------
```


www.it-ebooks.info

Then, we create a row with a discount percentage:

```
INSERT INTO Sales.InvoiceLineItem(InvoiceId, ProductId, Quantity,
                                  Cost, Discount, DiscountExplanation)
SELECT  (SELECT InvoiceId
         FROM Sales.Invoice
         WHERE InvoiceNumber = 'IE00000001'),
        (SELECT ProductId
         FROM Inventory.Product
         WHERE name = 'Cow Picture'),
        1,150,.45,'Customer purchased two, so I gave 45% off';
```

Checking the audit log this time:

```
SELECT * FROM Sales.InvoiceLineItemDiscountAudit;
```

Now, we see that a result has been logged:

```
InvoiceLineItemDiscountAudit     InvoiceId      InvoiceLineItemId      AuditTime
-----------------------------    ---------      -----------------      -----------------------
1                                1              2                      2012-02-18 23:03:05.823

SetByUserId                Quantity      Cost           Discount
--------------------       --------      --------       --------
DENALI-PC\AlienDrsql       1.00          150.0000       0.45

DiscountExplanation
---------------------------------------
Customer purchased two, so I gave 45% off
```

DML triggers make wonderful security devices for keeping an eye on what users do, because those triggers can be completely transparent to users *and* to programmers. The only catch is when an application layer isn't passing the security context through to the application. Sometimes, the application uses one common login, and there isn't any automatic way for the database code to determine which user is actually doing the DML operation. Most of the time when that is the case, you'll have already dealt with the problem using some method (such as just passing the username to the stored procedure, adding a column to the table to hold the user that is doing the operation, and so on).

DDL Triggers

DDL triggers let you watch what users do, but instead of watching what they do to data, you can watch what they do to the system. They let us protect and monitor changes to the server or database structure by firing when a user executes any DDL statement. The list of DDL statements you can monitor is quite long. (There are server-level events, such as creating and altering logins, as well as for the database, including creating and modifying tables, indexes, views, procedures, and so on. For a full list, check SQL Server Books Online in the "DDL Events Groups" topic.)

DDL triggers are of no value in protecting data values, because they don't fire for operations where data is changed or manipulated. They are, however, good for monitoring and preventing changes to the system, even by users who have the rights to do so. For example, consider the all-too-frequent case where the manager of the IT group has system administration powers on the database, though he or she can barely spell "SQL" (if this power wasn't granted, it would seem like a slight to the abilities and power of this manager). Now, let's assume that

www.it-ebooks.info

this manager is just pointing and clicking around the UI, and one click is to the wrong place, and all of a sudden, your customer table joins the choir invisible. Now, you have to restore from a backup and waste a day cleaning up the mess, while trying to figure out who dropped the table. (OK, so if you have constraints on your table, you can't actually drop it that easily. And yes, to be honest, most every DBA has dropped some object in a production database. That ends the honest part of this section.)

Preventing a DDL Action

With a simple DDL trigger, we can prevent the accidental drop of the table by trapping for the event and stopping it, or we can log who it was who dropped the table. In the first example, I will create a DDL trigger that will prevent any alterations to the schema without the user going in and manually disabling this trigger. It is a great safeguard to secure your objects from accidental change.

```
CREATE TRIGGER tr_server$allTableDDL_prevent --note, not a schema owned object
ON DATABASE
AFTER CREATE_TABLE, DROP_TABLE, ALTER_TABLE
AS
  BEGIN
    BEGIN TRY --note the following line will not wrap
        RAISERROR ('The trigger: tr_server$allTableDDL_prevent must be disabled
          before making any table modifications',16,1);
    END TRY
    --using the same old error handling
    BEGIN CATCH
          IF @@TRANCOUNT > 0
                  ROLLBACK TRANSACTION;

          THROW;
    END CATCH
END;
```

Now, we try to create a simple table:

```
CREATE TABLE dbo.testDDLTrigger --dbo for simplicity of example
(
 testDDLTriggerId int identity CONSTRAINT PKtest PRIMARY KEY
);
```

We get the following error message:

```
Msg 50000, Level 16, State 1, Procedure tr_server$allTableDDL_prevent, Line 7
The trigger: tr_server$allTableDDL_prevent must be disabled before making any table
modifications
```

Game over; player dumb. We could also log the error message, so we can see whether this happens often and send an e-mail to the support DBA team members so they can discover what is going on.

■ **Note** I wouldn't put the name of the trigger in the error message if this were a production application that had any external exposure. Otherwise, the hackers get everything they need to disable the trigger right in the error message.

www.it-ebooks.info

Recording a DDL Action

The second case, and just as useful, is to log DDL that is executed in a database so you can see what has been done. Although stopping DDL is something I usually do in a production database, logging changes is something I often do in a development environment—not that logging is never useful in a production environment; it just shouldn't be as necessary. Tables in the production system should be very stable and changed only in an organized manner, not just randomly by a user or a DBA. Sometimes, I will use DDL logging to catch things that are routine such as index changes, so I will know to watch the new indexes especially closely for a while to see whether they are valuable.

Let's look at creating a trigger similar to the one created in the preceding section. The difference is that this time we will have the trigger monitor DDL changes, not prevent them. First, let's drop the trigger created previously:

```
--Note: Slight change in syntax to drop DDL trigger, requires clause indicating
--where the objects are
DROP TRIGGER tr_server$allTableDDL_prevent ON DATABASE;
```

Now, we create a table to contain the history of changes to our table:

```
--first create a table to log to
CREATE TABLE dbo.TableChangeLog
(
        TableChangeLogId int NOT NULL IDENTITY
          CONSTRAINT pkTableChangeLog PRIMARY KEY (TableChangeLogId),
        ChangeTime datetime NOT NULL,
        UserName   sysname NOT NULL,
        Ddl        varchar(max) NOT NULL
);
--so we can get as much of the batch as possible
```

And we build another trigger to fire when a user creates, alters, or drops a table:

```
--not a schema bound object
CREATE TRIGGER tr_server$allTableDDL
ON DATABASE
AFTER CREATE_TABLE, DROP_TABLE, ALTER_TABLE
AS
  BEGIN
        SET NOCOUNT ON; --to avoid the rowcount messages
        SET ROWCOUNT 0; --in case the client has modified the rowcount

        BEGIN TRY

                --we get our data from the EVENT_INSTANCE XML stream
                INSERT INTO dbo.TableChangeLog (ChangeTime, UserName, Ddl)
                SELECT GETDATE(), USER,
EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)[1]',
        'nvarchar(max)');
  END TRY
  --using the same old error handling
  BEGIN CATCH
        IF @@TRANCOUNT > 0
          ROLLBACK TRANSACTION;
        THROW;
  END CATCH
```


www.it-ebooks.info

```
END;
```

Now, we run this to create the `dbo.test` table:

```
CREATE TABLE dbo.testDdlTrigger
(
  dbo.testDdlTriggerId int
);
DROP TABLE dbo.testDdlTrigger;
```

We check out the `TableChangeLog` data to see what has changed:

```
SELECT * FROM dbo.TableChangeLog;
```

This shows us our commands:

```
TableChangeLogId     ChangeTime                UserName     Ddl
----------------     ----------                --------     ---
1                    2012-02-18 23:16:51.753   dbo          CREATE TABLE dbo.testDdlTrigger
                                                            (
                                                            testDdlTriggerId int
                                                            )
2                    2011-02-18 23:16:51.793   dbo          DROP TABLE dbo.testDdlTrigger
```

Now, we can see what users have been up to in the database without them having to do anything special to cause it to happen (or without them even knowing, either).

■ **Tip** It's usually best when building production-quality applications not to have users dropping and creating tables, even with a good set of schema structures with which to work. DDL triggers give us the power to see what kind of activity is occurring. You can also use DDL triggers to prevent unwanted DDL from occurring, as I demonstrated earlier in this section as well.

Logging with Profiler

The last line of defense is the "security camera" approach, particularly if you are using a version that doesn't support database auditing. Just watch the activity on your server and make sure it looks legit. This is probably the only approach that has a chance to work with malicious (or stupid) programmers and DBAs. I include DBAs in here, because there's often no way to avoid giving a few of them system-administration powers. Hence, they might have access to parts of the data that they should never go to, unless there's a problem. They might stumble into data they shouldn't see, though as we discussed previously, we can use encryption to obfuscate this data. Unfortunately, even encryption won't necessarily stop a user with `sys_admin` rights.

Using Profiler, we can set up filters to look at certain events that we know shouldn't be happening, even when a DBA has access. For example, think back to our encryption example, where we stored the encryption password in the table. We could formulate a Profiler task to log only usage of this object. This log might be pretty small, especially if we filter out users who *should* be regularly accessing encrypted data. None of this is perfect, because any users who have admin rights to the Windows server and the database server could hide their activity with enough effort.

www.it-ebooks.info

I won't demonstrate building a server side trace, but you can use `sp_trace_create` and `sp_trace_setevent` to create a trace to watch for certain actions. Of course, profiler is way more than a security tool, and it's a tool you really need to learn to be a great SQL programmer. For a deep look at profiler, check out my good friend Brad McGehee's book *Mastering SQL Server Profiler*, published by Simple-Talk.

Best Practices

Security is always one of the most important tasks to consider when implementing a system. Storing data could be worse than not storing it, if it can be used for improper purposes.

- *Secure the server first*: Although this topic is outside the scope of this book, be certain that the server is secure. If a user can get access to your backup files and take them home, all the database security in the world won't help.
- *Grant rights to roles rather than users*: People come and people go, but the roles that they fulfill will be usually be around for a long time. By defining common roles, you can make adding a new user easy (possibly to replace another user). Just make the users a member of the same role, rather than adding rights directly to the user.
- *Use schemas to simplify security*: Because you can grant rights at a schema level, you can grant rights to `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and even `EXECUTE` everything within a schema. Even new objects that are added to the schema after the rights are granted are usable by the grantees.
- *Consider security using stored procedures*: Using stored procedures as the only way for a user to get access to the data presents the user with a nice interface to the data. If procedures are well named, you can also easily apply security to match up with the interfaces that use them.
- *Don't overuse the impersonation features*: `EXECUTE AS` is a blessing, and it has opened up a world of possibilities. It does, however, have a darker side because it can open up too much of a security hole without careful consideration of its use. Add a database with `TRUSTWORTHY` access set to on, and a procedure can be written to do anything on the server, which could be exploited as a big security hole by a devious programmer.
- *Encrypt sensitive data*: SQL Server has several means of encrypting data, and there are other methods available to do it off of the SQL Server box. Use it as much as necessary, but make sure not to store everything needed to decrypt the data with the encrypted data, in case someone gets hold of the data. Use transparent data encryption to secure important files from exploit if they fall into the wrong hands.
- *Use Profiler and DDL triggers to monitor system activity*: Sometimes, it's advantageous to keep an eye on user activity, and these tools give you the ability to do this in an easy manner.
- *Segregate security between environments*: Security in development environments will be very different. Take care not to end up with developers ending up with the same rights to production data as they have in development, because you use the same security script to create your development servers as you do in production. Developers generally should be given very few rights to production data to limit access to sensitive data.

www.it-ebooks.info

Summary

Security is a large topic, and understanding all the implications is way more than we covered in this chapter. I discussed some of the ways to secure your data inside a single SQL Server database. This isn't an easy subject, but it's far easier than dealing with securing the SQL Server. Luckily, usually in the database we're looking to protect ourselves from ordinary users, though doing a good job of encryption is a good barricade to keep most thieves at bay.

To provide this security, we discussed a range of topics for which we need to design security into our database usage:

- The basics of permissions-based security using SQL Server DDL statements and how this security works on SQL Server objects. This included using principals of several types: users, roles, and application roles, and then applying different security criteria to the base tables and columns in a database.
- Using coded objects to encapsulate statements that can limit the queries that users can execute. We discussed using several types of objects:
    - *Stored procedures and scalar functions*: Giving advanced usages to users without letting them know how they're doing it. Included in this section was how security works across database lines and server lines.
    - *Views and table-valued functions*: Used to break tables up in a simple manner, either row-or column-wise. The goal is to make security seamless, such that the users feel that only this database has the data to which they have rights.
- We looked at obfuscating data to make it too darn hard to view the data unless you specifically try to, generally by using encryption to make the data unreadable without a key.
- Next, we discussed watching users in a manner that that's analogous to how you might with a store security camera that watches all of the customers, looking for one to do something wrong. A few techniques we discussed were
    - *Using an audit trail*: Giving the user an audit of what goes on in given rows and columns in the database. This is the typical method when it comes to most data, because it's easy to give the users access to the lists of what has changed (and why, if the application asks for a reason with certain types of changes).
    - *DDL triggers*: Auditing users who have rights to create new objects in your databases or server to make sure they aren't doing anything out of the ordinary.
    - *Logging with a profiler*: The most silent of devices, it can be used to capture all moves made on the server.

Securing your servers against most common threats is that ordinary users won't go to great lengths to hack your database because getting caught can cause loss of employment. Hence, just setting up basic security is generally good enough for all but the really sensitive/valuable data (such as a database of credit card numbers linked with names and addresses of the card holders . . . not a good idea).

Of course, make sure that you understand that there is a lot more to security than just security on the database. The biggest task is limiting the people/processes that can even connect to the database to the correct set of users, and that means working with the administrators of your network, web sites, and applications to make sure to limit the threat surface as much as possible.

www.it-ebooks.info

CHAPTER 10

Table Structures and Indexing

A lot of us have jobs where we need to give people structure but that is different from controlling.

—Keith Miller

To me, the true beauty of the relational database engine comes from its declarative nature. As a programmer, I simply ask the engine a question, and it answers it. The questions I ask are usually pretty simple; just give me some data from a few tables, correlate it on some of the data, do a little math perhaps, and give me back these pieces of information (and naturally do it incredibly fast if you don't mind). Generally, the engine obliges with an answer extremely quickly. But how does it do it? If you thought it was magic, you would not be right. It is a lot of complex code implementing a massive amount of extremely complex algorithms that allow the engine to answer your questions in a timely manner. With every passing version of SQL Server, that code gets better at turning your relational request into a set of operations that gives you the answers you desire in remarkably small amounts of time. These operations will be shown to you on a query plan, which is a blueprint of the algorithms used to execute your query. I will use query plans often in this chapter and others to show you how your design choices can affect the way work gets done.

Our job as data-oriented designers and programmers is to assist the query optimizer (which takes your query and turns it into a plan of how to run the query), the query processor (which takes the plan and uses it to do the actual work), and the storage engine (which manages IO for the whole process) by first designing and implementing as close to the relational model as possible by normalizing your structures, using good set-based code (no cursors), following best practices with coding T-SQL, and so on. This is a design book, so I won't cover T-SQL coding, but it is a skill you should master. Consider Apress's *Beginning T-SQL 2012* by Kathi Kellenberger (Aunt Kathi!) and Scott Shaw or perhaps one of Itzik Ben-Gan's Inside SQL books on T-SQL for some deep learning on the subject. Once you have built your system correctly, the next step is to help out by adjusting the physical structures using indexing, filegroups, files, partitioning, and everything else you can do to adjust the physical layers to assist the optimizer deal with your commonly asked questions.

When it comes to tuning your database structures, you must maintain a balance between doing too much and too little. Indexing strategies are a great example of this. If you don't use indexes enough, searches will be slow, as the query processor could have to read every row of every table for every query (which, even if it seems fast on your machine, can cause the concurrency issues we will cover in the next chapter by forcing the query processor to lock a lot more resources than is necessary). Use too many indexes, and modifying data could take too long, as indexes have to be maintained. Balance is the key, kind of like matching the amount of fluid to the size of the glass so that you will never have to answer that annoying question about a glass that has half as much fluid as it can hold. (The answer is either that the glass is too large or the waitress needs to refill your glass immediately, depending on the situation.)

www.it-ebooks.info

Everything we have done so far has been centered on the idea that the quality of the data is the number one concern. Although this is still true, in this chapter, we are going to assume that we've done our job in the logical and implementation phases, so the data quality is covered. Slow and right is *always* better than fast and wrong (how would you like to get paid a week early, but only get half your money?), but the obvious goal of building a computer system is to do things right *and* fast. Everything we do for performance should affect only the performance of the system, not the data quality in any way.

We have technically added indexes in previous chapters as a side effect of adding primary key and unique constraints (in that a unique index is built by SQL Server to implement the uniqueness condition). In many cases, those indexes will turn out to be a lot of what you need to make normal queries run nicely, since the most common searches that people will do will be on identifying information. Of course, you will likely discover that some of the operations you are trying to achieve won't be nearly as fast as you hope. This is where physical tuning comes in, and at this point, you need to understand how tables are structured and consider organizing the physical structures.

The goal of this chapter is to provide a basic understanding of the types of things you can do with the physical database implementation, including the indexes that are available to you, how they work, and how to use them in an effective physical database strategy. This understanding relies on a base knowledge of the physical data structures on which we based these indexes—in other words, of how the data is structured in the physical SQL Server storage engine. In this chapter, I'll cover the following:

- *Physical database structure*: An overview of how the database and tables are stored. This acts mainly as foundation material for subsequent indexing discussion, but the discussion also highlights the importance of choosing and sizing your datatypes carefully.
- *Indexing*: A survey of the different types of indexes and their structure. I'll demonstrate many of the index settings and how these might be useful when developing your strategy, to correct any performance problems identified during optimization testing.
- *Index usage scenarios*: I'll discuss a few specialized cases of how to apply and use indexes.
- *Index Dynamic Management View queries*: In this section, I will introduce a couple of the dynamic management views that you can use to help determine what indexes you may need and to see which indexes have been useful in your system.

Once you understand the physical data structures, it will be a good bit easier to visualize what is occurring in the engine and then optimize data storage and access without affecting the correctness of the data. It's essential to the goals of database design and implementation that the physical storage not affect the physically implemented model. This is what Codd's eighth rule, also known as the Physical Data Independence rule—is about. As we discussed in Chapter 1, this rule states that the physical storage can be implemented in any manner as long as the users don't have to know about it. It also implies that if you change the physical storage, the users shouldn't be affected. The strategies we will cover should change the physical model but not the model that the users (people and code) know about. All we want to do is enhance performance, and understanding the way SQL Server stores data is an important step.

■ **Note** I am generally happy to treat a lot of the deeper internals of SQL Server as a mystery left to the engine to deal with. For a deeper explanation consider any of Kalen Delaney's books, where I go whenever I feel the pressing need to figure out why something that seems bizarre is occurring. The purpose of this chapter is to give you a basic feeling for what the structures are like, so you can visualize the solution to some problems and understand the basics of how to lay out your physical structures.

Some of the samples may not work 100% the same way on your computer, depending on our hardware situations, or changes to the optimizer from updates or service packs.

---

www.it-ebooks.info

Physical Database Structure

In SQL Server, databases are physically structured as several layers of containers that allow you to move parts of the data around to different disk drives for optimum access. As discussed in Chapter 1, a database is a collection of related data. At the logical level, it contains tables that have columns that contain data. At the physical level, databases are made up of *files*, where the data is physically stored. These files are basically just typical Microsoft Windows files, and they are logically grouped into *filegroups* that control where they are stored on a disk. Each file contains a number of *extents,* which are is an allocation of 64-KB in a database file that's made up of eight individual contiguous 8-KB *pages*. The page is the basic unit of data storage in SQL Server databases. Everything that's stored in SQL Server is stored on pages of several types—data, index, overflow and others—but these are the ones that are most important to you (I will list the others later in the section called "Extents and Pages"). The following sections describe each of these containers in more detail, so you understand the basics of how data is laid out on disk.

■ **Note** Because of the extreme variety of hardware possibilities and needs, it's impossible in a book on design to go into serious depth about how and where to place all your files in physical storage. I'll leave this task to the DBA-oriented books. For detailed information about choosing and setting up your hardware, check out `http://msdn.microsoft.com` or most any of Glenn Berry's writing. Glenn's blog (at `http://sqlserverperformance.wordpress.com/` at the time of this writing) contained a wealth of information about SQL Server hardware, particularly CPU changes.

Files and Filegroups

Figure 10-1 provides a high-level depiction of the objects used to organize the files (I'm ignoring logs in this chapter, because you don't have direct access to them).

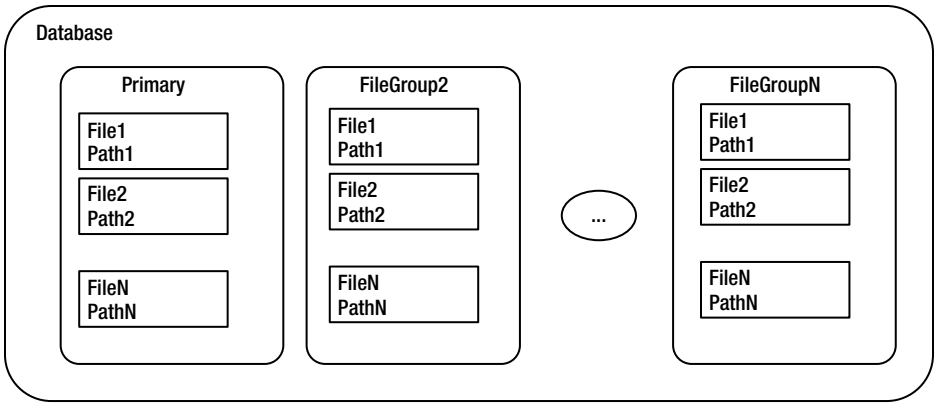


Figure 10-1. *Database storage organization*

At the top level of a SQL Server instance, we have the database. The database is comprised of one or more filegroups, which are logical groupings of one or more files. We can place different filegroups on different disk drives (hopefully on a different disk drive controller) to distribute the I/O load evenly across the available

www.it-ebooks.info

hardware. It's possible to have multiple files in the filegroup, in which case SQL Server allocates space across each file in the filegroup. For best performance, it's generally best to have no more files in a filegroup than you have physical CPUs (not including hyperthreading, though the rules with hyperthreading are changing as processors continue to improve faster than one could write books on the subject).

A filegroup contains one or more files, which are actual operating system files. Each database has at least one primary filegroup, whose files are called *primary files* (commonly suffixed as `.mdf`, although there's no requirement to give the files any particular names or extension). Each database can possibly have other secondary filegroups containing the secondary files (commonly suffixed as `.ndf`), which are in any other filegroups. Files may only be a part of a single filegroup. SQL Server proportionally fills files by allocating extents in each filegroup equally, so you should make all of the files the same size if possible. (There is also a file type for full-text indexing and a filegroup type we used in Chapter 7 for filestream types of data that I will largely ignore in this chapter as well. I will focus only on the core file types that you will use for implementing your structures.)

You control the placement of objects that store physical data pages at the filegroup level (code and metadata is always stored on the primary filegroup, along with all the system objects). New objects created are placed in the *default* filegroup, which is the `PRIMARY` filegroup (every database has one as part of the `CREATE DATABASE` statement, or the first file specified is set to primary) unless another filegroup is specified in any `CREATE <object>` commands. For example, to place an object in a filegroup other than the default, you need to specify the name of the filegroup using the `ON` clause of the table- or index-creation statement:

```
CREATE TABLE <tableName>
(…)    ON <fileGroupName>
```

This command assigns the table to the filegroup, but not to any particular file. Where in the files the object is created is strictly out of your control.

■ **Tip** If you want to move a table to a different filegroup, you can use the `MOVE TO` option of the `ALTER TABLE` statement if the table has a clustered index, or for a heap (a table without a clustered index, covered later in this chapter), create a clustered index on the object on the filegroup you want it and then drop it. For nonclustered indexes, use the `DROP_EXISTING` setting on the `CREATE INDEX` statement.

Use code like the following to create indexes and specify a filegroup:

```
CREATE INDEX <indexName> ON <tableName> (<columnList>) ON <filegroup>;
```

Use the following type of command (or use `ALTER TABLE`) to create constraints that in turn create indexes (`UNIQUE`, `PRIMARY KEY`):

```
CREATE TABLE <tableName>
(
    …
    <primaryKeyColumn> int CONSTRAINT PKTableName ON <fileGroup>
    …
);
```

For the most part, having just one filegroup and one file is the best practice for a large number of databases. If you are unsure if you need multiple filegroups, my advice is to build your database on a single filegroup and see if the data channel provided can handle the I/O volume (for the most part, I will avoid making too many such generalizations, as tuning is very much an art that requires knowledge of the actual load the server will be under). As activity increases and you build better hardware with multiple CPUs and multiple drive channels, you might place indexes on their own filegroup, or even place files of the same filegroup across different controllers.

www.it-ebooks.info

In the following example, I create a sample database with two filegroups, with the secondary filegroup having two files in it (I put this sample database in an SQL\Data folder in the root of the C drive to keep the example simple (and able to work on the types of drives that many of you will probably be testing my code on), but it is rarely a good practice to place your files on the C:\ drive when you have others available. I generally put a \sql directory on every drive and put everything SQL in that directory in subfolders to keep things consistent over all of our servers. Put the files wherever works best for you.):

```
CREATE DATABASE demonstrateFilegroups ON
PRIMARY ( NAME = Primary1, FILENAME = 'c:\sql\data\demonstrateFilegroups_primary.mdf',
          SIZE = 10MB),
FILEGROUP SECONDARY
        ( NAME = Secondary1, FILENAME = 'c:\sql\data\demonstrateFilegroups_secondary1.ndf',
          SIZE = 10MB),
        ( NAME = Secondary2, FILENAME = 'c:\sql\data\demonstrateFilegroups_secondary2.ndf',
          SIZE = 10MB)
LOG ON ( NAME = Log1,FILENAME = 'c:\sql\log\demonstrateFilegroups_log.ldf', SIZE = 10MB);
```

You can define other file settings, such as minimum and maximum sizes and growth. The values you assign depend on what hardware you have. For growth, you can set a FILEGROWTH parameter that allows you to grow the file by a certain size or percentage of the current size, and a MAXSIZE parameter, so the file cannot just fill up existing disk space. For example, if you wanted the file to start at 1GB and grow in chunks of 100MB up to 2GB, you could specify the following:

```
CREATE DATABASE demonstrateFileGrowth ON
PRIMARY ( NAME = Primary1,FILENAME = 'c:\sql\data\demonstrateFileGrowth_primary.mdf',
          SIZE = 1GB, FILEGROWTH=100MB, MAXSIZE=2GB)
LOG ON ( NAME = Log1,FILENAME = 'c:\sqll\data\demonstrateFileGrowth_log.ldf', SIZE = 10MB);
```

The growth settings are fine for smaller systems, but it's usually better to make the files large enough so that there's no need for them to grow. File growth can be slow and cause ugly bottlenecks when OLTP traffic is trying to use a file that's growing. When SQL Server is running on a desktop operating system like Windows XP or greater (and at this point you probably ought to be using something greater like Windows 7—(or presumably Windows 8 or 9 depending on when you are reading this) or on a server operating system such as Windows Server 2003 or greater (again, it is 2012, so emphasize "or greater"), you can improve things by using "instant" file allocation (though only for data files). Instead of initializing the files, the space on disk can simply be allocated and not written to immediately. To use this capability, the system account cannot be LocalSystem, and the user account that the SQL Server runs under must have SE_MANAGE_VOLUME_NAME Windows permissions. Even with the existence of instant file allocation, it's still going to be better to have some idea of what size data you will have and allocate space proactively, as you then have cordoned off the space ahead of time: no one else can take it from you, and you won't fail when the file tries to grow and there isn't enough space. In either event, the DBA staff should be on top of the situation and make sure that you don't run out of space.

You can query the sys.filegroups catalog view to view the files in the newly created database:

```
USE demonstrateFilegroups;
GO
SELECT CASE WHEN fg.name IS NULL
            then CONCAT('OTHER-',df.type_desc COLLATE DATABASE_DEFAULT)
                  ELSE fg.name END AS file_group,
        df.name AS file_logical_name,
        df.physical_name AS physical_file_name
FROM    sys.filegroups fg
          RIGHT JOIN sys.database_files df
              ON fg.data_space_id = df.data_space_id;
```


www.it-ebooks.info

This returns the following results:

```
file_group          file_logical_name       physical_file_name
---------           ----------------        ------------------------------------------------
PRIMARY             Primary1                c:\sql\data\demonstrateFilegroups_primary.mdf
OTHER-LOG           Log1                    c:\sql\log\demonstrateFilegroups_log.ldf
SECONDARY           Secondary1              c:\sql\data\demonstrateFilegroups_secondary1.ndf
SECONDARY           Secondary2              c:\sql\data\demonstrateFilegroups_secondary2.ndf
```

The LOG file isn't technically part of a filegroup, so I used a right outer join to the database files and gave it a default filegroup name of OTHER plus the type of file to make the results include all files in the database. You may also notice a couple other interesting things in the code. First, the CONCAT function is new to SQL Server 2012 to add strings together.. Second is the COLLATE database_default. The strings in the system functions are in the collation of the server, while the literal OTHER- is in the database collation. If the server doesn't match the database, this query would fail.

There's a lot more information than just names in the catalog views I've referenced already in this chapter. If you are new to the catalog views, dig in and learn them. There is a wealth of information in those views that will be invaluable to you when looking at systems to see how they are set up and to determine how to tune them.

■ **Tip** An interesting feature of filegroups is that you can back up and restore them individually. If you need to restore and back up a single table for any reason, placing it in its own filegroup can achieve this.

These databases won't be used anymore, so if you created them, just drop them if you desire:

```
USE MASTER;
GO
DROP DATABASE demonstrateFileGroups;
GO
DROP DATABASE demonstrateFileGrowth;
```

Extents and Pages

As shown in Figure 10-2, files are further broken down into a number of *extents*, each consisting of eight separate 8-KB pages where tables, indexes, and so on are physically stored. SQL Server only allocates space in a database to extents. When files grow, you will notice that the size of files will be incremented only in 64-KB increments.

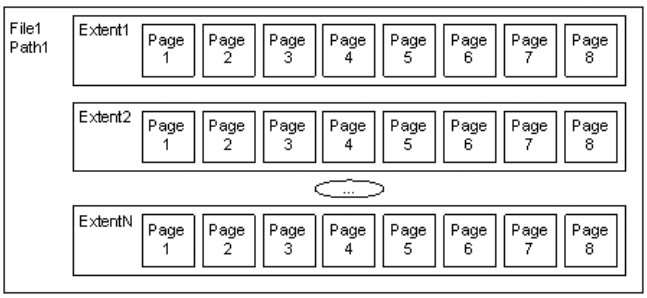


Figure 10-2. *Files and extents*

www.it-ebooks.info

Each extent in turn has eight pages that hold one specific type of data each:

- *Data*: Table data.
- *Index*: Index data.
- *Overflow data:* Used when a row is greater than 8,060 bytes or for `varchar(max)`, `varbinary(max)`, text, or image values.
- *Allocation map*: Information about the allocation of extents.
- *Page free space*: Information about what different pages are allocated for.
- *Index allocation*: Information about extents used for table or index data.
- *Bulk changed map*: Extents modified by a bulk `INSERT` operation.
- *Differential changed map*: Extents that have changed since the last database backup command. This is used to support differential backups.

In larger databases, most extents will contain just one type of page, but in smaller databases, SQL Server can place any kind of page in the same extent. When all data is of the same type, it's known as a *uniform* extent. When pages are of various types, it's referred to as a *mixed* extent.

SQL Server places all table data in pages, with a header that contains metadata about the page (object ID of the owner, type of page, and so on), as well as the rows of data, which I'll cover later in this chapter. At the end of the page are the offset values that tell the relational engine where the rows start.

Figure 10-3 shows a typical data page from a table. The header of the page contains identification values such as the page number, the object ID of the object the data is for, compression information, and so on. The data rows hold the actual data. Finally, there's an allocation block that has the offsets/pointers to the row data.

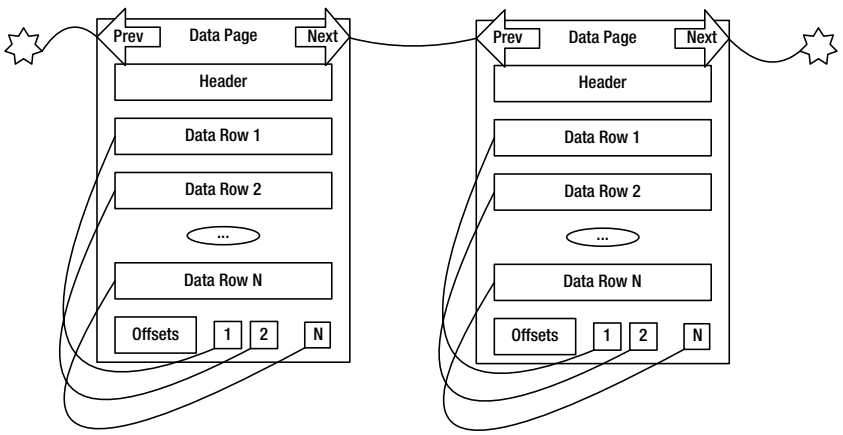


Figure 10-3. *Data pages*

www.it-ebooks.info

Figure 10-3 shows that there are pointers from the next to the previous rows. These pointers are only used when pages are ordered, such as in the pages of an index. Heap objects (tables with no clustered index) are not ordered. I will cover this a bit later in the "Index Types" section.

The other kind of page that is frequently used that you need to understand is the *overflow page*. It is used to hold row data that won't fit on the basic 8,060-byte page. There are two reasons an overflow page is used:

- The combined length of all data in a row grows beyond 8,060 bytes. In versions of SQL Server prior to 2000, this would cause an error. In versions after this, data goes on an overflow page automatically, allowing you to have virtually unlimited row sizes.
- By setting the `sp_tableoption` setting on a table for `large value types out of row` to `1`, all the `(max)` and `XML` datatype values are immediately stored out of row on an overflow page. If you set it to `0`, SQL Server tries to place all data on the main page in the row structure, as long as it fits into the 8,060-byte row. The default is `0`, because this is typically the best setting when the typical values are short enough to fit on a single page.

For example, Figure 10-4 depicts the type of situation that might occur for a table that has the `large value types out of row` set to `1`. Here, `Data Row 1` has two pointers to a `varbinary(max)` columns: one that spans two pages and another that spans only a single page. Using all of the data in `Data Row 1` will now require up to four reads (depending on where the actual page gets stored in the physical structures), making data access far slower than if all of the data were on a single page. This kind of performance problem can be easy to overlook, but on occasion, overflow pages will really drag down your performance, especially when other programmers use `SELECT *` on tables where they don't really need all of the data.

The overflow pages are linked lists that can accommodate up to 2GB of storage in a single column. Generally speaking, it isn't really a very good idea to store 2GB in a single column (or even a row), but the ability to do so is available if needed.

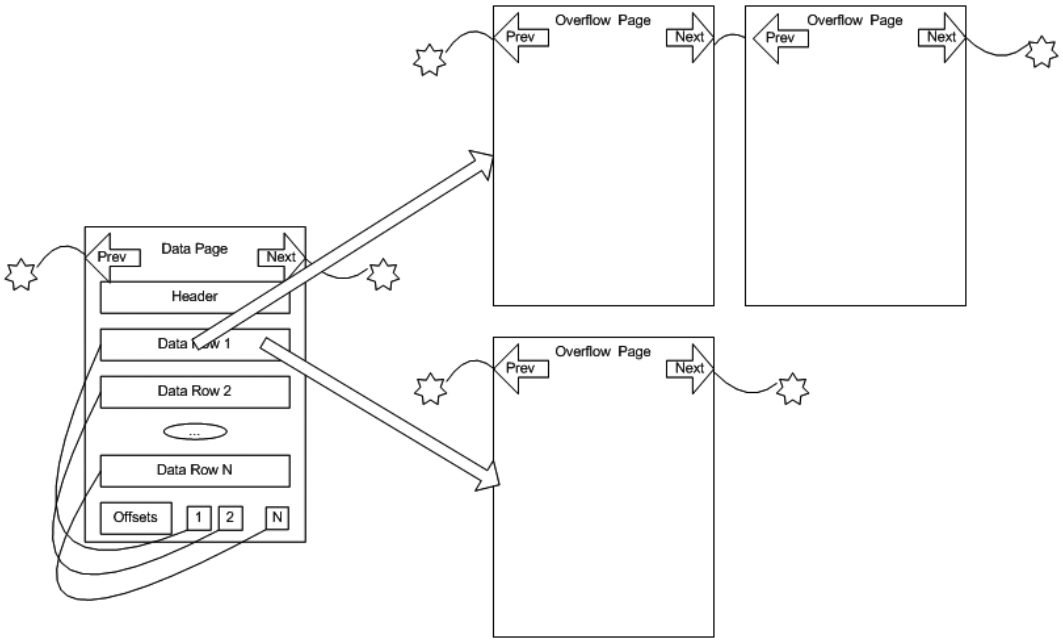


Figure 10-4. *Sample overflow pages*

www.it-ebooks.info

Understand that storing large values that are placed off of the main page will be far more costly when you need these values than if all of the data can be placed in the same data page. On the other hand, if you seldom use the data in your queries, placing them off the page can give you a much smaller footprint for the important data, requiring far less disk access on average. It is a balance that you need to take care with, as you can imagine how costly a table scan of columns that are on the overflow pages is going to be. Not only will you have to read extra pages, you have to be redirected to the overflow page for every row that's overflowed.

Be careful when allowing data to overflow the page. It's guaranteed to make your processing more costly, especially if you include the data that's stored on the overflow page in your queries—for example, if you use the dreaded `SELECT *` regularly in production code! It's important to choose your datatypes correctly to minimize the size of the data row to include only frequently needed values. If you frequently need a large value, keep it in row; otherwise, consider placing it off row or even create two tables and join them together as needed.

■ **Tip** The need to access overflow pages is just one of the reasons to avoid using `SELECT * FROM <tablename>`–type queries in your production code, but it is an important one. Too often, you get data that you don't intend to use, and when that data is located off the main data page, performance could suffer tremendously, and, in most cases, needlessly.

Data on Pages

When you get down to the row level, the data is laid out with metadata, fixed length fields, and variable length fields, as shown in Figure 10-5. (Note that this is a generalization, and the storage engine does a lot of stuff to the data for optimization, especially when you enable compression.)

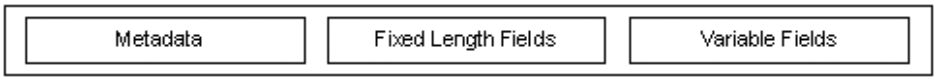


Figure 10-5. *Data row*

The metadata describes the row, gives information about the variable length fields, and so on. Generally speaking, since data is dealt with by the query processor at the page level, even if only a single row is needed, data can be accessed very rapidly no matter the exact physical representation.

■ **Note** I use the term "column" when discussing logical SQL objects such as tables and indexes, but when discussing the physical table implementation, "field" is the proper term. Remember from Chapter 1 that a field is a physical location within a record.

The maximum amount of data that can be placed on a single page (including overhead from variable fields) is 8,060 bytes. As illustrated in Figure 10-4, when a data row grows larger than 8,060 bytes, the data in variable length columns was can spill out onto an overflow page. A 16-byte pointer is left on the original page and points to the page where the overflow data is placed.

www.it-ebooks.info

Page Splits

When inserting or updating rows, SQL Server might have to rearrange the data on the pages due to the pages being filled up. Such rearranging can be a particularly costly operation. Consider the situation from our example shown in Figure 10-6, assuming that only three values can fit on a page.

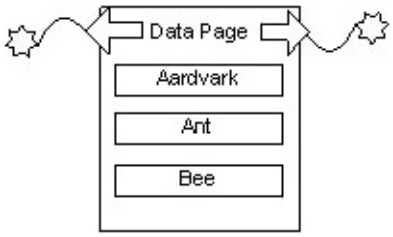


Figure 10-6. *Sample data page before page split*

Say we want to add the value `Bear` to the page. If that value won't fit onto the page, the page will need to be reorganized. Pages that need to be split are split into two, generally with 50 percent of the data on one page, and 50 percent on the other (there are usually more than three values on a real page). Once the page is split and its values are reinserted, the new pages would end up looking something like Figure 10-7.

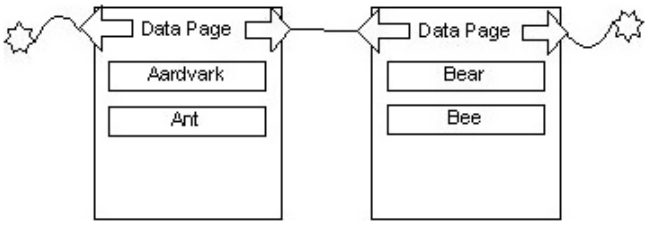


Figure 10-7. *Sample data page after page split*

Page splits are awfully costly operations and can be terrible for performance, because after the page split, data won't be located on successive physical pages. This condition is commonly known as *fragmentation*. Page splits occur in a normal system and are simply a part of adding data to your table. However, they can occur extremely rapidly and seriously degrade performance if you are not careful. Understanding the effect that page splits can have on your data and indexes is important as you tune performance on tables that have large numbers of inserts or updates.

To tune your tables and indexes to help minimize page splits, you can use the `FILL FACTOR` of the index. When you build or rebuild an index or a table (using `ALTER TABLE <tablename> REBUILD`, a command that was new in SQL Server 2008), the fill factor indicates how much space is left on each page for future data. If you are inserting random values all over the structures, a common situation that occurs when you use a nonsequential `uniqueidentifier` for a primary key, you will want to leave adequate space on each page to cover the expected numbers of rows that will be created in the future. During a page split, the data page is always split approximately fifty-fifty, and it is left half empty on each page, and even worse, the structure is becoming, as mentioned, fragmented.

www.it-ebooks.info

Let's jump ahead a bit: one of the good things about using a monotonously increasing value for a clustered index is that page splits over the entire index are greatly decreased. The table grows only on one end of the index, and while the index does need to be rebuilt occasionally using `ALTER INDEX REORGANIZE` or `ALTER INDEX REBUILD`, you don't end up with page splits all over the table. And since the new value that won't fit on the page comes at the end of the table, instead of a split, another page can be added to the chain. In the "Index Dynamic Management Views" section later, I will provide a query that will help you to know when to rebuild the index due to fragmentation.

Compression

The reason that the formats for pages and rows are generalized in the previous sections is that in SQL Server 2008, Microsoft implemented compression of data at the row and page levels. In versions prior to SQL Server 2005 SP2, all data was stored on a page in a raw format. However, in SQL Server 2005 SP2, Microsoft introduced datatype-level compression, which allowed data of the `decimal` datatype to be stored in a variable length field, also referred to as `vardecimal`. For SQL Server 2005, datatype compression was set using the `sp_tableoption` procedure with a setting of `'vardecimal storage format'`.

In SQL Server 2008, the concept of compression was extended even further to all of the fixed length datatypes, including `int`, `char`, and `float`. Basically, you can allow SQL Server to save space by storing your data like it was a variable-sized type, yet in usage, the data will appear and behave like a fixed length type. Then in 2008R2, the feature was expanded yet again to include Unicode values. In Appendix A, I will note how compression will affect each datatype individually.

For example, if you stored the value of 100 in an `int` column, SQL Server needn't use all 32 bits; it can store the value 100 the same amount of space as a `tinyint`. So instead of taking a full 32 bits, SQL Server can simply use 8 bits (1 byte). Another case is when you use a fixed length type like `char(30)` column but store only two characters; 28 characters could be saved, and the data padded as it is used. There is an overhead of 2 bytes per variable length column (or 4 bits if the size of the column is less than 8 bytes). Note that compression is only available in the Enterprise Edition.

This datatype-level compression is referred to as *row compression*, where each row in the table will be compressed as datatypes allow, shrinking the size on disk, but not making any major changes to the page structure. In Appendix B, I will indicate how each of the datatypes is affected by row compression, or for a list that may show any recent changes, you can check SQL Server Books Online for the topic of "Row Compression Implementation." Row compression is a very interesting thing for many databases that use lots of fixed length data (for example, integers, especially for surrogate keys).

SQL Server also includes an additional compression capability called *page compression*. With page compression, first the data is compressed in the same manner as row compression, and then, the storage engine does a couple of interesting things to compress the data on a page:

- *Prefix compression*: Looks for repeated values in a value (like `'0000001'` and compresses the prefix to something like `6-0` (six zeros)
- *Dictionary compression*: For all values on the page, the storage engine looks for duplication, stores the duplicated value once, and then stores pointers on the data pages where the duplicated values originally resided.

You can apply data compression to your tables and indexes with the `CREATE TABLE`, `ALTER TABLE`, `CREATE INDEX`, and `ALTER INDEX` syntaxes. As an example, I will create a simple table, called `test`, and enable page compression on the table, row compression on a clustered index, and page compression on another index. (This code will work only on an Enterprise installation, or if you are using Developer Edition for testing.)

www.it-ebooks.info

```
USE tempdb;
GO
CREATE TABLE testCompression
(
    testCompressionId int NOT NULL,
    value int NOT NULL
);
WITH (DATA_COMPRESSION = ROW) -- PAGE or NONE
    ALTER TABLE testCompression REBUILD WITH (DATA_COMPRESSION = PAGE);

CREATE CLUSTERED INDEX XTestCompression_value
    ON testCompression (value) WITH ( DATA_COMPRESSION = ROW );

ALTER INDEX XTestCompression_value
    ON testCompression REBUILD WITH ( DATA_COMPRESSION = PAGE );
```

■ **Note** The syntax of the `CREATE INDEX` command allows for compression of the partitions of an index in different manners. I mention partitioning in the next section of the chapter. For full syntax, refer to SQL Server Books Online.

Giving advice on whether to use compression is not really possible without knowing the factors that surround your actual situation. One tool you should use is the system procedure—`sp_estimate_data_compression_savings`—to check existing data to see just how compressed the data in the table or index would be after applying compression, but it won't tell you how the compression will positively or negatively affect your performance. There are trade-offs to any sorts of compression. CPU utilization will go up in most cases, because instead of directly using the data right from the page, the query processor will have to translate the values from the compressed format into the uncompressed format that SQL Server will use. On the other hand, if you have a lot of data that would benefit from compression, you could possibly lower your I/O enough to make doing so worth the cost. Frankly, with CPU power growing by leaps and bounds with multiple-core scenarios these days and I/O still the most difficult to tune, compression could definitely be a great thing for many systems. However, I suggest testing with and without compression before applying in your production systems.

Partitioning

The last general physical structure concept that I will introduce is *partitioning*. Partitioning allows you to break a table (or index) into multiple physical structures by breaking them into more manageable chunks. Partitioning can allow SQL Server to scan data from different processes, enhancing opportunities for parallelism. SQL Server 7.0 and 2000 had partitioned views, where you would define a view and a set of tables, with each serving as a partition of the data. If you have properly defined (and trusted) constraints, SQL Server would use the `WHERE` clause to know which of the tables referenced in the view would have to be scanned in response to a given query. The data in the view was also editable like in a normal table. One thing you still can do with partitioned views is to build distributed partitioned views, which reference tables on different servers.

In SQL Server 2005, you could begin to define partitioning as part of the table structure. Instead of making a physical table for each partition, you define, at the DDL level, the different partitions of the table. Internally, the table is broken into the partitions based on a scheme that you set up. Note, too, that this feature is only included in the Enterprise Edition.

At query time, SQL Server can then dynamically scan only the partitions that need to be searched, based on the criteria in the `WHERE` clause of the query being executed. I am not going to describe partitioning too much, but I felt that it needed a mention in this edition of this book as a tool at your disposal with which to

www.it-ebooks.info

tune your databases, particularly if they are very large or very active. For deeper coverage, I would suggest you consider on of Kalen Delaney's *SQL Server Internals* books. They are the gold standard in understanding the internals of SQL Server.

I will, however, present the following basic example of partitioning. Use whatever database you desire. I used `tempdb` for the data and `AdventureWorks2012` for the sample data on my test machine and included the `USE` statement in the code download. The example is that of a sales order table. I will partition the sales into three regions based on the order date. One region is for sales before 2006, another for sales between 2006 and 2007, and the last for 2007 and later. The first step is to create a partitioning function. You must base the function on a list of values, where the `VALUES` clause sets up partitions that the rows will fall into based on the `smalldatetime` values that are presented to it, for example:

```
CREATE PARTITION FUNCTION PartitionFunction$dates (smalldatetime)
AS RANGE LEFT FOR VALUES ('20060101','20070101');
    --set based on recent version of
    --AdventureWorks2012.Sales.SalesOrderHeader table to show
    --partition utilization
```

Specifying the function as `RANGE LEFT` says that the values in the comma-delimited list should be considered the boundary on the side listed. So in this case, the ranges would be as follows:

- `value <= '20060101'`
- `value > '20060101'` and `value <= '20070101'`
- `value > '20070101'`

Specifying the function as `RANGE RIGHT` would have meant that the values lie to the right of the values listed, in the case of our ranges, for example:

- `value < '20060101'`
- `value >= '20060101'` and `value < '20070101'`
- `value >= '20070101'`

Next, use that partition function to create a partitioning scheme:

```
CREATE PARTITION SCHEME PartitonScheme$dates
    AS PARTITION PartitionFunction$dates ALL to ( [PRIMARY] );
```

which will let you know:

```
Partition scheme 'PartitonScheme$dates' has been created successfully. 'PRIMARY' is marked as
the next used filegroup in partition scheme 'PartitonScheme$dates'.
```

With the `CREATE PARTITION SCHEME` command, you can place each of the partitions you previously defined on a specific filegroup. I placed them all on the same filegroup for clarity and ease, but in practice, you usually want them on different filegroups, depending on the purpose of the partitioning. For example, if you were partitioning just to keep the often-active data in a smaller structure, placing all partitions on the same filegroup might be fine. But if you want to improve parallelism or be able to just back up one partition with a filegroup backup, you would want to place your partitions on different filegroups.

www.it-ebooks.info

Next, you can apply the partitioning to a new table. You'll need a clustered index involving the partition key. You apply the partitioning to that index. Following is the statement to create the partitioned table:

```
CREATE TABLE dbo.salesOrder
(
    salesOrderId      int NOT NULL,
    customerId        int NOT NULL,
    orderAmount       decimal(10,2) NOT NULL,
    orderDate         smalldatetime NOT NULL,
    CONSTRAINT PKsalesOrder PRIMARY KEY NONCLUSTERED (salesOrderId)
                                                      ON [Primary],
    CONSTRAINT AKsalesOrder UNIQUE CLUSTERED (salesOrderId, orderDate)
) on PartitonScheme$dates (orderDate);
```

Next, load some data from the `AdventureWorks2012.Sales.SalesOrderHeader` table to make looking at the metadata more interesting. You can do that using an `INSERT` statement such as the following:

```
INSERT  INTO  dbo.salesOrder(salesOrderId, customerId, orderAmount, orderDate)
SELECT  SalesOrderID, CustomerID, TotalDue, OrderDate
FROM    AdventureWorks2012.Sales.SalesOrderHeader;
```

You can see what partition each row falls in using the `$partition` function. You suffix the `$partition` function with the partition function name and the name of the partition key (or a partition value) to see what partition a row's values are in, for example:

```
SELECT *, $partition.PartitionFunction$dates(orderDate) AS partiton
FROM   dbo.salesOrder;
```

You can also view the partitions that are set up through the `sys.partitions` catalog view. The following query displays the partitions for our newly created table:

```
SELECT    partitions.partition_number, partitions.index_id,
          partitions.rows, indexes.name, indexes.type_desc
FROM      sys.partitions AS partitions
              JOIN sys.indexes AS indexes
                  on indexes.object_id = partitions.object_id
                      AND indexes.index_id = partitions.index_id
WHERE     partitions.object_id = object_id('dbo.salesOrder');
```

This will return the following:

```
partition_number     index_id      rows         name             type_desc
----------------     --------      -----        ------------     -------------
1                    1             1424         AKsalesOrder     CLUSTERED
2                    1             3720         AKsalesOrder     CLUSTERED
3                    1             26321        AKsalesOrder     CLUSTERED
1                    2             31465        PKsalesOrder     NONCLUSTERED
```


www.it-ebooks.info

Partitioning is not a general purpose tool that should be used on every table, which is one of the reasons why it is only included in Enterprise Edition. However, partitioning can solve a good number of problems for you, if need be:

- *Performance*: If you only ever need the past month of data out of a table with three years' worth of data, you can create partitions of the data where the current data is on a partition and the previous data is on a different partition.
- *Rolling windows*: You can remove data from the table by dropping a partition, so as time passes, you add partitions for new data and remove partitions for older data (or move to a different archive table).
- *Maintainence*: Some maintainance can be done at the partition level rather than the entire table, so once partition data is read-only, you may not need to maintain any longer. Some caveats do apply (you cannot rebuild a partitioned index online, for example.)

Indexes Overview

Indexes allow the SQL Server engine to perform fast, targeted data retrieval rather than simply scanning though the entire table. A well-placed index can speed up data retrieval by orders of magnitude, while a haphazard approach to indexing can actually have the opposite effect when creating, updating, or deleting data.

Indexing your data effectively requires a sound knowledge of how that data will change over time, the sort of questions that will be asked of it, and the volume of data that you expect to be dealing with. Unfortunately, this is what makes any topic about physical tuning so challenging. To index effectively, you almost need the psychic ability to fortell the future of your exact data usage patterns. Nothing in life is free, and the creation and maintenance of indexes can be costly. When deciding to (or not to) use an index to improve the performance of one query, you have to consider the effect on the overall performance of the system.

In the upcoming sections, I'll do the following:

- Introduce the basic structure of an index.
- Discuss the two fundamental types of indexes and how their structure determines the structure of the table.
- Demonstrate basic index usage, introducing you to the basic syntax and usage of indexes.
- Show you how to determine whether SQL Server is likely to use your index and how to see if SQL Server has used your index.

If you are producing a product for sale that uses SQL Server as the backend, indexes are truly going to be something that you could let your customers manage unless you can truly effectively constrain how users will use your product. For example, if you sell a product that manages customers, and your basic expectation is that they will have around 1,000 customers, what happens if one wants to use it with 100,000 customers? Do you not take their money? Of course you do, but what about performance? Hardware improvements generally cannot even give linear improvement in performance. So if you get hardware that is 100 times "faster," you would be extremely fortunate to get close to 100 times improvement. However, adding a simple index can provide 100,000 times improvement that may not even make a difference at all on the smaller data set. (This is not to pooh pooh the value of faster hardware at all. Just that situationally you get far greater gain from writing better code than you will from just throwing hardware at the problem. The ideal situation is adequate hardware and excellent code, naturally).

www.it-ebooks.info

Basic Index Structure

An index is an object that SQL Server can maintain to optimize access to the physical data in a table. You can build an index on one or more columns of a table. In essence, an index in SQL Server works on the same principle as the index of a book. It organizes the data from the column (or columns) of data in a manner that's conducive to fast, efficient searching, so you can find a row or set of rows without looking at the entire table. It provides a means to jump quickly to a specific piece of data, rather than just starting on page one each time you search the table and scanning through until you find what you're looking for. Even worse, unless SQL Server knows *exactly* how many rows it is looking for, it has no way to know if it can stop scanning data when one row had been found. Also, like the index of a book, an index is a separate entity from the actual table (or chapters) being indexed.

As an example, consider that you have a completely unordered list of employees and their details. If you had to search this list for persons named "Davidson", you would have to look at every single name on every single page. Soon after trying this, you would immediately start trying to devise some better manner of searching. On first pass, you would probably sort the list alphabetically. But what happens if you needed to search for an employee by an employee identification number? Well, you would spend a bunch of time searching through the list sorted by last name for the employee number. Eventually, you could create a list of last names and the pages you could find them on and another list with the employee numbers and their pages. Following this pattern, you would build indexes for any other type of search you'd regularly perform on the list. Of course, SQL Server can page through the phone book one name at a time in such a manner that, if you need to do it occasionally, it isn't such a bad thing, but looking at two or three names per search is always more efficient than two or three hundred, much less two or three million.

Now, consider this in terms of a table like an `Employee` table. You might execute a query such as the following:

```
SELECT  LastName, <EmployeeDetails>
FROM    Employee
WHERE   LastName = 'Davidson';
```

In the absence of an index to rapidly search, SQL Server will perform a scan of the data in the entire table (referred to as a table scan) on the `Employee` table, looking for rows that satisfy the query predicate. A full table scan generally won't cause you too many problems with small tables, but it can cause poor performance for large tables with many pages of data, much as it would if you had to manually look through 20 values versus 2,000. Of course, when you have a light load, like on your development box, you probably won't be able to discern the difference between a seek and a scan (or even hundreds of scans). Only when you are experiencing a reasonably heavy load will the difference be noticed.

If we instead created an index on the `LastName` column, the index would sort the `LastName` rows in a logical fashion (in ascending alphabetical order by default) and the database engine can move directly to rows where the last name is Davidson and retrieve the required data quickly and efficiently. And even if there are ten people with the last name of Davidson, SQL Server knows to stop when it hits `'Davidtown'`.

Of course, as you might imagine, the engineer types who invented the concept of indexing and searching data structures don't simply make lists to search through. Instead, indexes are implemented using what is known as a *balanced tree* (B-tree) structure. The index is made up of index pages structured, again, much like an index of a book or a phone book. Each index page contains the first value in a range and a pointer to the next lower page in the index. The last level in the index is referred to as the *leaf page*, which contains the actual data values that are being indexed, plus either the data for the row or pointers to the data. This allows the query processor to go directly to the data it is searching for by checking only a few pages, even when there are a millions of values in the index.

Figure 10-8 shows an example of the type of B-tree that SQL Server uses for indexes. Each of the outer rectangles is an 8K index page, just as we discussed earlier. The three values—A, J, and P—are the *index keys* in this top-level page of the index. The index page has as many index keys as is possible. To decide which path to follow to reach the lower level of the index, we have to decide if the value requested is between two of the keys: A to I, J to P, or greater than P. For example, say the value we want to find in the index happens to be `I`. We go to

www.it-ebooks.info

the first page in the index. The database determines that `I` doesn't come after `J`, so it follows the `A` pointer to the next index page. Here, it determines that `I` comes after `C` and `G`, so it follows the `G` pointer to the leaf page.

Each of these pages is 8KB in size. Depending on the size of the key (determined by summing the data lengths of the columns in the key, up to a maximum of 900 bytes), it's possible to have anywhere from 8 entries to over 1,000 on a single page. The more keys you can fit on a page, the greater the number of pages you can have on each level of the index. The more pages are linked from each level to the next, the fewer numbers of steps from the top page of the index to reach the leaf.

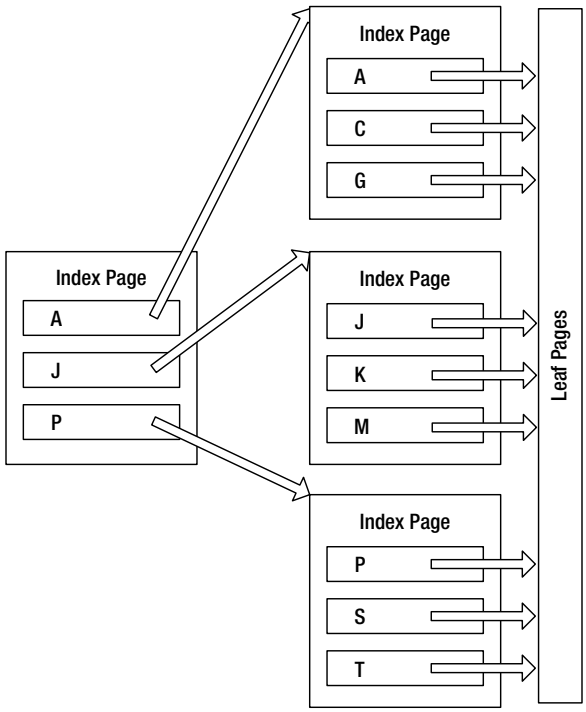


Figure 10-8. *Basic index structure*

B-tree indexes are extremely efficient, because for an index that stores only 500 different values on a page—a reasonable number for a typical index of an integer—it has 500 pointers on the next level in the index, and the second level has 500 pages with 500 values each. That makes 250,000 different pointers on that level, and the next level has up to 250,000 * 500 pointers. That's 125,000,000 different values in just a three-level index. Change that to a 100-byte key, do the math, and you will see why smaller keys are better! Obviously, there's overhead to each index key, and this is just a rough estimation of the number of levels in the index.

www.it-ebooks.info

Another idea that's mentioned occasionally is how well balanced the tree is. If the tree is perfectly balanced, every index page would have exactly the same number of keys on it. Once the index has lots of data on one end, or data gets moved around on it for insertions or deletions, the tree becomes ragged, with one end having one level, and another many levels. This is why you have to do some basic maintenance on the indexes, something I have mentioned already.

Index Types

How indexes are structured internally is based on the existence (or nonexistence) of a clustered index. For the nonleaf pages of an index, everything is the same for all indexes. However, at the leaf node, the indexes get quite different—and the type of index used plays a large part in how the data in a table is physically organized.

There are two different types of relational indexes:

- *Clustered*: This type orders the physical table in the order of the index.
- *Nonclustered*: These are completely separate structures that simply speed access.

In the upcoming sections, I'll discuss how the different types of indexes affect the table structure and which is best in which situation.

Clustered Indexes

A clustered index physically orders the pages of the data in the table. The leaf pages of the clustered indexes are the data pages of the table. Each of the data pages is then linked to the next page in a doubly linked list to provide ordered scanning. The leaf pages of the clustered index are the actual data pages. In other words, the records in the physical structure are sorted according to the fields that correspond to the columns used in the index. Tables with clustered indexes are referred to as *clustered tables*.

The key of a clustered index is referred to as the *clustering key*, and this key will have additional uses that will be mentioned later in this chapter. For clustered indexes that aren't defined as unique, each record has a 4-byte value (commonly known as an *uniquifier*) added to each value in the index where duplicate values exist. For example, if the values were A, B, and C, you would be fine. But, if you added another value B, the values internally would be A, B + `4ByteValue`, B + `Different4ByteValue`, and C. Clearly, it is not optimal to get stuck with 4 bytes on top of the other value you are dealing with in every level of the index, so in general, you should try to use the clustered index on a set of columns where the values are unique.

Figure 10-9 shows, at a high level, what a clustered index might look like for a table of animal names. (Note that this is just a partial example, there would likely be more second-level pages for Horse and Python at a minimum.)

You can have only one clustered index on a table, because the table cannot be ordered in more than one direction. (Remember this; it is one of the most fun interview questions. Answering anything other than "one clustered index per table" leads to a fun line of followup questioning.)

A good real-world example of a clustered index would be a set of old-fashioned encyclopedias. Each letter is a level of the index, and on each page, there is another level that denotes the things you can find on each page (e.g., Office–Officer). Then. each topic is the leaf level of the index. The encyclopedia is are clustered on the topics in these books, just as the example was clustered on the name of the animal. In essence, the entire book is a table of information in clustered ordered. And indexes can be partitioned as well. The encylopedias are partitioned by letter into multiple books.

Now, consider a dictionary. Why are the words sorted, rather than just having a separate index with the words not in order? I presume that at least part of the reason is to let the readers scan through words they don't know exactly how to spell, checking the definition to see if the word matches what they expect. SQL Server does something like this when you do a search. For example, back in Figure 10-9, if you were looking for a cat named

www.it-ebooks.info

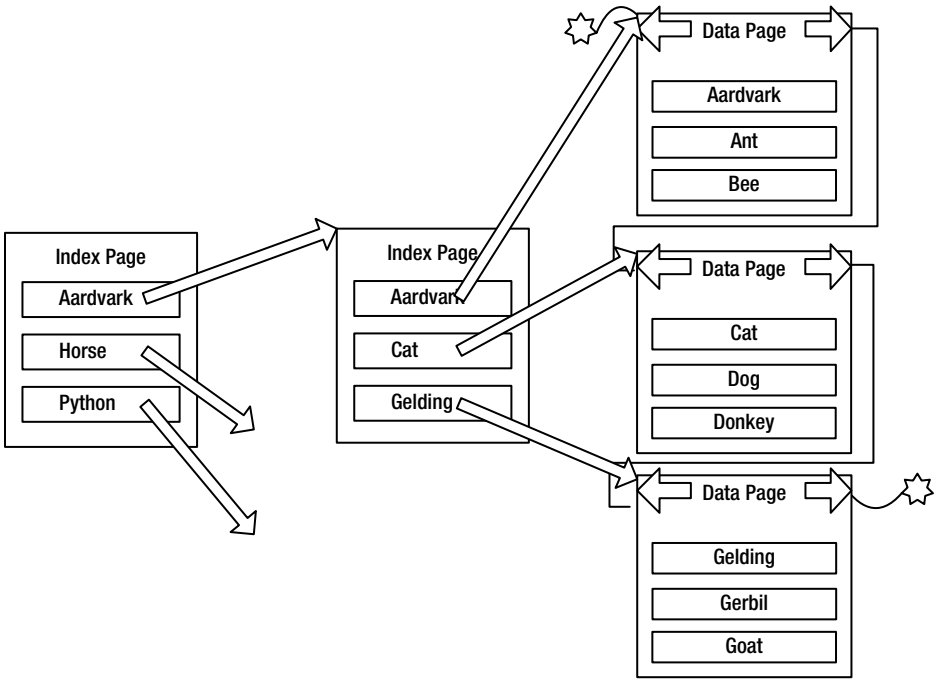


Figure 10-9. *Clustered index example*

George, you could use the clustered index to find rows where `animal = 'Cat'`, then scan the data pages for the matching pages for any rows where `name = 'George'`.

I must caution you that although it's true, physically speaking, that tables are stored in the order of the clustered index; logically speaking, tables must be thought of as having no order. (I know I promised to not mention this again back in Chapter 1, but it really is an important thing to remember.) This lack of order is a fundamental truth of relational programming: *you aren't required to get back data in the same order when you run the same query twice.* The ordering of the physical data can be used by the query processor to enhance your performance, but during intermediate processing, the data can be moved around in any manner that results in faster processing the answer to your query. It's true that you do almost always get the same rows back in the same order, mostly because the optimizer is almost always going to put together the same plan every time the same query is executed under the same conditions. However, load up the server with many requests, and the order of the data might change so SQL Server can best use its resources, regardless of the data's order in the structures. SQL Server can choose to return data to us in any order that's fastest for it. If disk drives are busy in part of a table and it can fetch a different part, it will. If order matters, use an `ORDER BY` clause to make sure that data is returned as you want.

Nonclustered Indexes

Nonclustered index structures are fully independent of the underlying table. Where a clustered index is like a dictionary with the index physically linked to the table (since the leaf pages of the index are a part of the table), nonclustered indexes are more like indexes in a textbook. A nonclustered index is completely separate from the data, and on the leaf page, there are pointers to go to the data pages much like the index of a book contains page numbers.

www.it-ebooks.info

Each leaf page in a nonclustered index contains some form of pointer to the rows on the data page. The pointer from the index to a data row is known as a *row locator*. Exactly how the row locator of a nonclustered indexes is structured is based on whether or not the underlying table has a clustered index.

In this section, I will first show an abstract representation of the nonclustered index and then show the differences between the implementation of a nonclustered index when you do and do not also have a clustered index. At an abstract level, all nonclustered indexes follow the form shown in Figure 10-10.

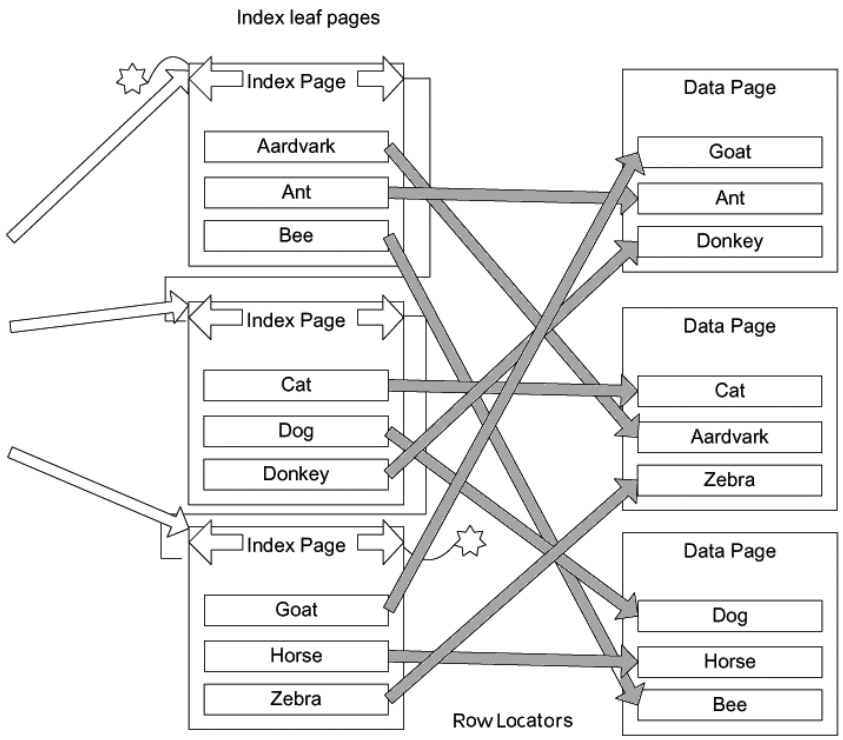


Figure 10-10. *Sample nonclustered index*

The major difference between the two possibilities comes down to the row locator being different based on whether the underlying table has a clustered index. There are two different types of pointer that will be used:

- *Tables with a clustered index*: Clustering key
- *Tables without a clustered index*: Pointer to physical location of the data

In the next two sections, I'll explain these in more detail. New in 2012 is a new type of index that I will briefly mention called a *columnstore* index. Rather than store index data per row, they store data by column. They are typical indexes, in that they don't change the structure of the data, but they do make the table read only. It is a very exciting feature for data warehouse types of queries, but in relational, OLTP databases, there isn't really a use for columnstore indexes. In Chapter 14, a columnstore index will be used in an example for the data warehouse overview, but that will be the extent of discussion on the subject. They are structured quite differently from classic B-tree indices, so if you do get to building a data warehouse (or if you are also involved with building dimensional databases), you will want to understand how they work and how they are structured as they will likely provide your star schema queries with a tremendous performance increase.

www.it-ebooks.info

■ **Tip** You can place nonclustered indexes on a different filegroup than the data pages to maximize the use of your disk subsystem in parallel. Note that the filegroup you place the indexes on ought to be on a different controller channel than the table; otherwise, it's likely that there will be minimal or no gain.

Nonclustered Indexes on Clustered Tables

When a clustered index exists on the table, the row locator for the leaf node of any nonclustered index is the clustering key from the clustered index. In Figure 10-10, the structure on the right side represents the clustered index, and on the left, the nonclustered index. To find a value, you start at the leaf node of the index and traverse the leaf pages. The result of the index traversal is the clustering key, which you then use to traverse the clustered index to reach the data, as shown in Figure 10-11.

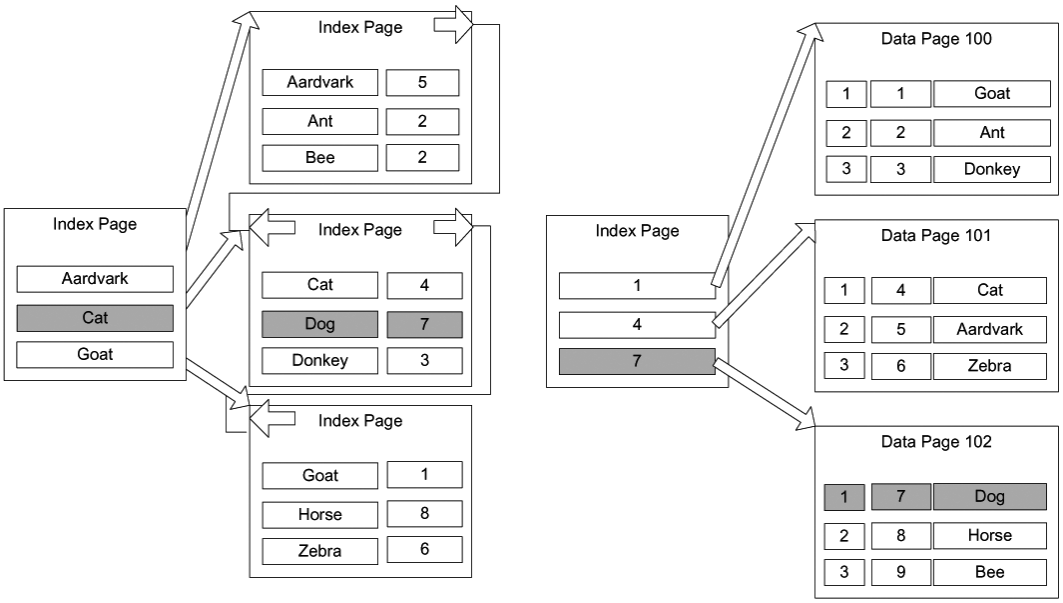


Figure 10-11. *Nonclustered index on a clustered table*

The overhead of the operation I've just described is minimal as long as you keep your clustering key optimal and the index maintained. While having to scan two indexes is more work than just having a pointer to the physical location you have to think of the overall picture. Overall, it's better than having direct pointers to the table, because only minimal reorganization is required for any modification of the values in the table. Consider if you had to maintain a book index manually. If you used the book page as the way to get to an index value, if had to add a page to the book in the middle you would have to update all of the page numbers. But if all of the topics were ordered alphabetically, and you just pointed to the topic name adding a topic would be easy.

The same is true for SQL Server and the structures can be changed thousands of times a second or more. Since there is very little hardware-based information lingering in the structure, data movement is easy for the query processor, and maintaining indexes is an easy operation. Early versions of SQL Server used physical location pointers, and this led to all manners of corruption in our indexes and tables. And let's face it, the people

www.it-ebooks.info

with better understanding of such things also tell us that when the size of the clustering key is adequately small, this method is remarkably faster overall than having pointers directly to the table.

The primary benefit of the key structure becomes more obvious when we talk about modification operations. Because the clustering key is the same regardless of physical location, only the lowest level of the clustered index need know where the physical data is. Add to this that the data is organized sequentially, and the overhead of modifying indexes is significantly lowered making all of the data modification operations far faster. Of course, this benefit is only true if the clustering key rarely, or never, changes. Therefore, the general suggestion is to make the clustering key a small nonchanging value, such as an identity column (but the advice section is still a few pages away).

Nonclustered Indexes on a Heap

If a table does not have a clustered index, the table is physically referred to as a *heap*. One definition of a heap is "a group of things placed or thrown one on top of the other." This is a great way to explain what happens in a table when you have no clustered index: SQL Server simply puts every new row on the end of the last page for the table. Once that page is filled up, it puts a data on the next page or a new page as needed.

When building a nonclustered index on a heap, the row locator is a pointer to the physical page and row that contains the row. As an example, take the example structure from the previous section with a nonclustered index on the `name` column of an `animal` table, represented in Figure 10-12.

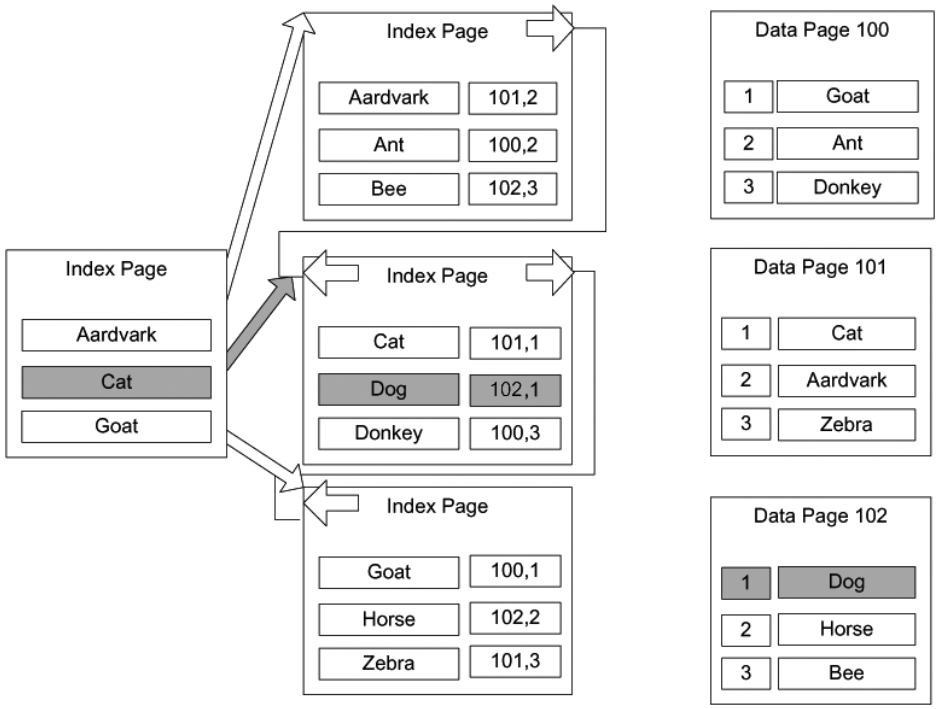


Figure 10-12. *Nonclustered index on a heap*

www.it-ebooks.info

If you want to find the row where `name ='Dog'`, you first find the path through the index from the top-level page to the leaf page. Once you get to the leaf page, you get a pointer to the page that has a row with the value, in this case Page 102, Row 1. This pointer consists of the page location and the record number on the page to find the row values (the pages are numbered from 0, and the offset is numbered from 1). The most important fact about this pointer is that it points directly to the row on the page that has the values you're looking for. The pointer for a table with a clustered index (a clustered table) is different, and this distinction is important to understand because it affects how well the different types of indexes perform.

To avoid the types of physical corruption issues that, as I mentioned in the previous section, can occur when you are constantly managing pointers and physical locations, heaps use a very simple method of keeping the row pointers from getting corrupted. Instead of reordering pages, or changing pointers if the page must be split, it moves rows to a different page and a *forwarding pointer* is left to point to the new page where the data is now. So if the row where `name ='Dog'` had moved (for example, due to a large `varchar(3000)` column being updated from a data length of 10 to 3,000), you might end up with following situation to extend the number of steps required to pick up the data. In Figure 10-13, a forwarding pointer is illustrated.

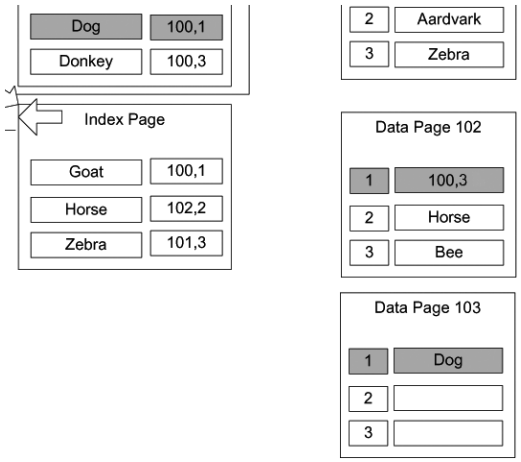


Figure 10-13. *Fowarding pointer*

All existing indexes that have the old pointer simply go to the old page and follow the forwarding pointer on that page to the new location of the data. If you are using heaps, which should be rare, it is important to be careful with your structures to make sure that data should rarely be moved around within a heap. For example, you have to be careful if you're often updating data to a larger value in a variable length column that's used as an index key, it's possible that a row may be moved to a different page. This adds another step to finding the data, and if the data is moved to a page on a different extent, another read to the database. This forwarding pointer is immediately followed when scanning the table, causing possible horrible performance over time if it's not managed.

Space is not reused in the heap without rebuilding the table (by selecting into another table, adding a clustered index temporarily, or in 2008, using the `ALTER TABLE` command with the `REBUILD` option). In the last section of this chapter on Indexing Dynamic Management View queries, I will provide a query that will give you information on the structure of your index, including the count of forwarding pointers in your table.

www.it-ebooks.info

Basics of Index Creation

The basic syntax for creating an index is as follows:

```
CREATE [UNIQUE] INDEX [CLUSTERED | NONCLUSTERED] <indexName>
    ON <tableName> (<columnList>);
```

As you can see, you can specify either a clustered or nonclustered index, with nonclustered being the default type. (Recall that the default index created for a primary key constrant is a clustered index.) Each type of index can be unique or nonunique. If you specify that your index must be unique, every row in the indexed column must have a different value—no duplicate entries are accepted, just like one of the uniqueness constraints.

The `<columnList>` is a comma-delimited list of columns in the table. Each column can be specified either in ascending (`ASC`) or descending (`DESC`) order for each column, with ascending being the default. SQL Server can traverse the index in either direction for searches, so the direction is generally only important when you have multiple columns in the index.

For example, the following statement creates an index called `XtableA_column1AndColumn2` on `column1` and `column2` of `tableA`, with ascending order for `column1` and descending order for `column2`:

```
CREATE INDEX XtableA_column1AndColumn2 ON tableA (column1, column2 DESC);
```

Let's take a look at a more detailed example. First, we need to create a base table. Use whatever database you desire. I used `tempdb` on my test machine and included the `USE` statement in the code download.

```
CREATE SCHEMA produce;
GO
CREATE TABLE produce.vegetable
(
    --PK constraint defaults to clustered
    vegetableId int NOT NULL CONSTRAINT PKproduce_vegetable PRIMARY KEY,
    name varchar(15) NOT NULL
                    CONSTRAINT AKproduce_vegetable_name UNIQUE,

    color varchar(10) NOT NULL,
    consistency varchar(10) NOT NULL,
    filler char(4000) DEFAULT (REPLICATE('a', 4000)) NOT NULL
);
```

■ **Note** I included a huge column in the produce table to exacerbate the issues with indexing the table in lieu of having tons of rows to work with. This causes the data to be spread among several data pages, and as such, forces some queries to perform more like a table with a lot more data. I will not return this filler column or acknowledge it much in the chapter, but this is its purpose. Obviously, creating a filler column is not a performance tuning best practice, but it is a common trick to force a table's data pages to take up more space in a demonstration.

Now, we create two single-column nonclustered indexes on the `color` and `consistency` columns, respectively:

```
CREATE INDEX Xproduce_vegetable_color ON produce.vegetable(color);
CREATE INDEX Xproduce_vegetable_consistency ON produce.vegetable(consistency);
```


www.it-ebooks.info

Then, we create a unique composite index on the vegetableID and color columns. We make this index unique, not to guarantee uniqueness of the values in the columns but because the values in vegetableId must be unique because it's part of the PRIMARY KEY constraint. Making this unique signals to the optimizer that the values in the index are unique (note that this index is probably not very useful in reality but is created to demonstrate a unique index that isn't a constraint).

```
CREATE UNIQUE INDEX Xproduce_vegetable_vegetableId_color
       ON produce.vegetable(vegetableId, color);
```

Finally, we add some test data:

```
INSERT  INTO produce.vegetable(vegetableId, name, color, consistency)
VALUES  (1,'carrot','orange','crunchy'), (2,'broccoli','green','leafy'),
        (3,'mushroom','brown','squishy'), (4,'pea','green','squishy'),
        (5,'asparagus','green','crunchy'), (6,'sprouts','green','leafy'),
        (7,'lettuce','green','leafy'),( 8,'brussels sprout','green','leafy'),
        (9,'spinach','green','leafy'), (10,'pumpkin','orange','solid'),
        (11,'cucumber','green','solid'), (12,'bell pepper','green','solid'),
        (13,'squash','yellow','squishy'), (14,'canteloupe','orange','squishy'),
        (15,'onion','white','solid'), (16,'garlic','white','solid');
```

To see the indexes on the table, we check the following query:

```
SELECT  name, type_desc, is_unique
FROM    sys.indexes
WHERE   OBJECT_ID('produce.vegetable') = object_id;
```

This returns the following results:

```
name                                type_desc          is_unique
----------------------------------- -------------      ---------
PKproduce_vegetable                 CLUSTERED          1
AKproduce_vegetable_name            NONCLUSTERED       1
Xproduce_vegetable_color            NONCLUSTERED       0
Xproduce_vegetable_consistency      NONCLUSTERED       0
Xproduce_vegetable_vegetableId_color NONCLUSTERED      1
```

One thing to remind you here is that PRIMARY KEY and UNIQUE constraints were implemented behind the scenes using indexes. The PK constraint is, by default, implemented using a clustered index and the UNIQUE constraint via a nonclustered index. As the primary key is generally chosen to be an optimally small value, it tends to make a nice clustering key.

■ **Note** Foreign key constraints aren't automatically implemented using an index, though indexes on migrated foreign key columns are often useful for performance reasons. I'll return to this topic later when I discuss the relationship between foreign keys and indexes.

---

www.it-ebooks.info

The remaining entries in the output show the three nonclustered indexes that we explicitly created and that the last index was implemented as unique since it included the unique key values as one of the columns. Before moving on, briefly note that to drop an index, use the `DROP INDEX` statement, like this one to drop the `Xproduce_vegetable_consistency` index we just created:

```
DROP INDEX Xproduce_vegetable_consistency ON produce.vegetable;
```

One last thing I want to mention on the basic index creation is a feature that was new to 2008. That feature is the ability to create *filtered indexes*. By including a `WHERE` clause in the `CREATE INDEX` statement, you can restrict the index such that the only values that will be included in the leaf nodes of the index are from rows that meet the `where` clause. We built a filtered index back in Chapter 8 when I introduced selective uniqueness, and I will mention them again in this chapter to show how to optimize for certain `WHERE` clauses.

The options I have shown so far are clearly not all of the options for indexes in SQL Server, nor are these the only types of indexes available. For example, there are options to place indexes on different filegroups from tables. Also at your disposal are filestream data (mentioned in Chapter 8), data compression, setting the maximum degree of parallelism to use with an index, locking (page or row locks), rebuilding, and several other features. There are also XML, spatial index types, and as previously mentioned, columnstore indexes. This book focuses specifically on relational databases, so relational indexes are really all that I am covering in any depth.

Basic Index Usage Patterns

In this section, I'll look at some of the basic usage patterns of the different index types, as well as how to see the use of the index within a query plan:

- *Clustered indexes*: I'll discuss the choices you need to make when choosing the columns in which to put the clustered index.
- *Nonclustered indexes*: After the clustered index is applied, you need to decide where to apply nonclustered indexes.
- *Unique indexes*: I'll look at why it's important to use unique indexes as frequently as possible.

All plans that I present will be obtained using the `SET SHOWPLAN_TEXT ON` statement. When you're doing this locally, it can be easier to use the graphical showplan from Management Studio. However, when you need to post the plan or include it in a document, use one of the `SET SHOWPLAN_TEXT` commands. You can read about this more in SQL Server Books Online. Note that using `SET SHOWPLAN_TEXT` (or the other versions of `SET SHOWPLAN` that are available, such as `SET SHOWPLAN_XML`), do not actually execute the statement/batch; rather, they show the estimated plan. If you need to execute the statement (like to get some dynamic SQL statements to execute to see the plan), you can use `SET STATISTICS PROFILE ON` to get the plan and some other pertinent information about what has been executed. Each of these session settings will need to be turned `OFF` explicitly once you have finished, or they will continue executing returning plans where you don't want so.

For example, say you execute the following query:

```
SET SHOWPLAN_TEXT ON;
GO
SELECT  *
FROM    produce.vegetable;
GO
SET SHOWPLAN_TEXT OFF;
GO
```


www.it-ebooks.info

Running these statements echoes the query as a single column result set of `StmtText` and then returns another with the same column name that displays the plan:

```
|--Clustered Index Scan(OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]))
```

Although this is the best way to communicate the plan in text (for example, to post to the MSDN/TechNet or any other forums to get help on why your query is so slow), it is not the richest or easiest experience. In SSMS, click the Query menu and choose Display Estimated Execution Plan (Ctrl+L); you'll see the plan in a more interesting way, as shown in Figure 10-14. Or by choosing Include Actual Execution Plan, you can see exactly what SQL Server did (which is analogous to `SET STATISTICS PROFILE ON`).

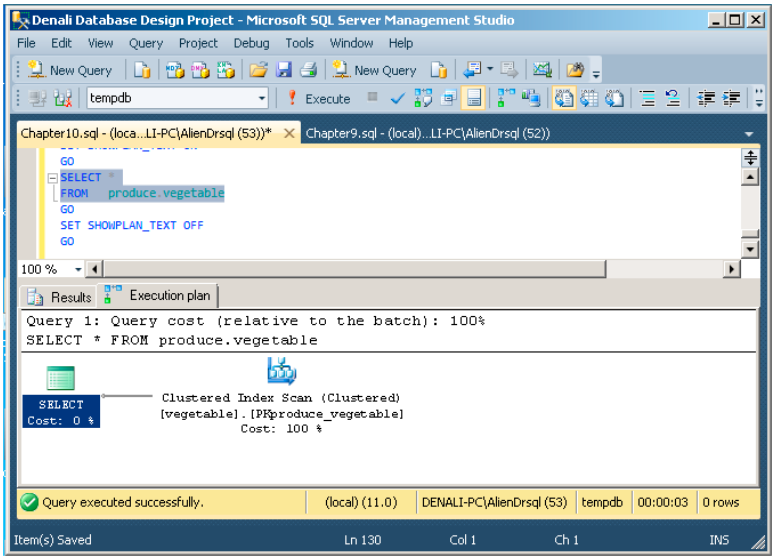


Figure 10-14. *Plan display in Management Studio*

Before breaking down the different index types, we need to start out by introducing a few terms need to be introduced:

- *Scan*: This refers to an unordered search, where SQL Server scans the leaf pages of the index looking for a value. Generally speaking, all leaf pages would be considered in the process.
- *Seek*: This refers to an ordered search, in that the index pages are used to go to a certain point in the index and then a scan is done on a range of values. For a unique index, this would always return a single value.
- *Lookup*: The clustered index is used to look up a value for the nonclustered index.

www.it-ebooks.info

Each of these types of operations is very important in understanding how an index is used in a query. Seeks are generally best; scans are less desirable. Lookups aren't necessarily bad, but too many lookups can be indicative of needing some tuning (often using covering indexes, a topic we will cover in a bit.)

Using Clustered Indexes

The column you use for the clustered index will (because the key is used as the row locator), become a part of every index for your table, so it has heavy implications for all indexes. Because of this, for a typical OLTP system, a very common practice is to choose a surrogate key value, often the primary key of the table, since the surrogate can be kept very small.

Using the surrogate key as the clustering key is usually a great decision, not only because is it a small key (most often, the datatype is an integer that requires only 4 bytes or possibly less using compression), but because it's always a unique value. As mentioned earlier, a nonunique clustering key has a 4-byte uniquifier tacked onto its value when keys are not unique. It also helps the optimizer that an index has only unique values, because it knows immediately that for an equality operator, either `1` or `0` values will match. Because the surrogate key is often used in joins, it's helpful to have smaller keys for the primary key.

■ **Caution** Using a GUID for a surrogate key is becoming the vogue these days, but be careful. GUIDs are 16 bytes wide, which is a fairly large amount of space, but that is really the least of the problem. They are random values, in that they generally aren't monotonically increasing, and a new GUID could sort anywhere in a list of other GUIDs.

Clustering on a random value is generally troublesome for inserts, because if you don't leave spaces on each page for new rows, you are likely to have page splitting. If you have a very active system, the constant page splitting can destroy your system, and the opposite effect, to leave lots of empty space, can be just as painful, as you make reads far less effective. The only way to make GUIDs a reasonably acceptable type is to use the `NEWSEQUENTIALID()` function (or one of your own) to build sequential GUIDS, but it only works with unique identifier columns in a default constraint. Seldom will the person architecting a solution that is based on GUID surrogates want to be tied down to using a default constraint to generate surrogate values. The ability to generate GUIDs from anywhere and ensure their uniqueness is part of the lure of the siren call of the 16-byte value. For SQL Server 2012, the use of the sequence object to generate guaranteed unique values could be used in lieu of GUIDs.

The clustered index won't always be used for the surrogate key or even the primary key. Other possible uses can fall under the following types:

- *Range queries*: Having all the data in order usually makes sense when there's data that you often need to get a range, such as from A to F. An example where this can make sense is for the child rows in a parent child relationship.
- *Data that's always accessed sequentially*: Obviously, if the data needs to be accessed in a given order, having the data already sorted in that order will significantly improve performance.
- *Queries that return large result sets*: This point will make more sense once I cover nonclustered indexes, but for now, note that having the data on the leaf index page saves overhead.

www.it-ebooks.info

The choice of how to pick the clustered index depends on a couple factors, such as how many other indexes will be derived from this index, how big the key for the index will be, and how often the value will change. When a clustered index value changes, every index on the table must also be touched and changed, and if the value can grow larger, well, then we might be talking page splits. This goes back to understanding the users of your data and testing the heck out of the system to verify that your index choices don't hurt overall performance more than they help. Speeding up one query by using one clustering key could hurt all queries that use the nonclustered indexes, especially if you chose a large key for the clustered index.

Frankly, in an OLTP setting, in all but the most unusual cases, I stick with a surrogate key for my clustering key, usually one of the integer types or sometimes even the `uniqueidentifier` (GUID) type (ideally the sequential type). I use the surrogate key because so many of the queries you do for modification (the general goal of the OLTP system) will access the data via the primary key. You then just have to optimize retrievals, which should also be of generally small numbers of rows, and doing so is usually pretty easy.

Another thing that is good about using the clustered index on a monotonically increasing value is that page splits over the entire index are greatly decreased. The table grows only on one end of the index, and while it does need to be rebuilt occasionally using `ALTER INDEX REORGANIZE` or `ALTER INDEX REBUILD`, you don't end up with page splits all over the table. You can decide which to do by using the criteria stated by SQL Server Books Online. By looking in the dynamic management view `sys.dm_db_index_physical_stats`, you can use `REBUILD` on indexes with greater than 30% fragmentation and use `REORGANIZE` otherwise. Now, let's look at an example of a clustered index in use. If you have a clustered index on a table, instead of `Table Scan`, you'll see a `Clustered Index Scan` in the plan:

```
SELECT  *
FROM    produce.vegetable;
```

The plan for this query is as follows:

```
|--Clustered Index Scan(OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]))
```

If you query on a value of the clustered index key, the scan will likely change to a seek. Although a scan touches all the data pages, a clustered index seek uses the index structure to find a starting place for the scan and knows just how far to scan. For a unique index with an equality operator, a seek would be used to touch one page in each level of the index to find (or not find) a single value on a single data page, for example:

```
SELECT  *
FROM    produce.vegetable
WHERE   vegetableId = 4;
```

The plan for this query now does a seek:

```
|--Clustered Index Seek(OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]),
      SEEK:([tempdb].[produce].[vegetable].[vegetableId]=
            CONVERT_IMPLICIT(int,[@1],0)) ORDERED FORWARD)
```

Note the `CONVERT_IMPLICIT` of the `@1` value. This shows the query is being parameterized for the plan, and the variable is cast to an integer type. In this case, you're seeking in the clustered index based on the `SEEK` predicate of `vegetableId = 4`. SQL Server will create a reusable plan by default for simple queries. Any queries that are executed with the same exact format, and a simple integer value would use the same plan. You can let SQL Server paramaterize more complex queries as well. (For more information, look up "Simple

www.it-ebooks.info

Paramaterization and Forced Parameterization" in Books Online.) Increasing the complexity, now, we search for two rows:

```
SELECT  *
FROM    produce.vegetable
WHERE   vegetableId IN (1,4);
```

And in this case, pretty much the same plan is used, except the seek criteria now has an OR in it:

```
|--Clustered Index Seek(OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]),
      SEEK:([tempdb].[produce].[vegetable].[vegetableId]=(1)
            OR [tempdb].[produce].[vegetable].[vegetableId]=(4)) ORDERED FORWARD)
```

Note that it did not create a parameterized plan this time, but a fixed one with literals for 1 and 4. Note that this plan will be executed as two separate seek operations. If you turn on SET STATISTICS IO before running the query:

```
SET STATISTICS IO ON;
GO
SELECT  *
FROM    produce.vegetable
WHERE   vegetableId IN (1,4);
Go
SET STATISTICS IO OFF;
```

You will see that it did two "scans", which using STATISTICS IO generally means any operation that probes the table, so a seek or scan would show up the same (note the lob values are for large objects such as (max) types):

```
Table 'vegetable'. Scan count 2, logical reads 4, physical reads 0, read-ahead reads 0,
lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

But whether any given query uses a seek or scan, or even two seeks, can be a pretty complex question. Why it is so complex will become clearer over the rest of the chapter, and it will become instantly clearer how useful a clustered index seek is in the next section, "Using Nonclustered Indexes."

A final consideration you need to think about is how the index will be used. There are two types of index usages, equality and inequality. In a query such as the previous ones, we have done as the following:

```
SELECT  *
FROM    produce.vegetable
WHERE   vegetableId = 4;
```

Or even the queries with the IN Boolean expression, these have been equality operators, where the values that were being searched for were known at compile time. For inequality searches, you will look for a value greater than some other value, or between values. So for this query, it has the following plan:

```
|--Clustered Index Seek(OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]),
      SEEK:([tempdb].[produce].[vegetable].[vegetableId]=
            CONVERT_IMPLICIT(int,[@1],0)) ORDERED FORWARD)
```


www.it-ebooks.info

Changing this to an inequality search, such as

```
SELECT  *
FROM    produce.vegetable
WHERE   vegetableId > 4;
```

Now, you will get the following plan:

```
|--Clustered Index Seek(OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]),
      SEEK:([tempdb].[produce].[vegetable].[vegetableId] > (4)) ORDERED FORWARD)
```

Note that, this time, even though we did a fairly simple query, it did not automatically parameterize. This is because the plan can change drastically based on the value passed.

Using Nonclustered Indexes

After you have made the ever-important choice of what to use for the clustered index, all other indexes will be nonclustered. In this section, I will cover nonclustered indexes in the following areas:

- General considerations
- Composite index considerations
- Nonclustered indexes with clustered tables
- Nonclustered indexes on heaps

General Considerations

We generally know that indexes are needed because queries are slow. Lack of indexes is clearly not the only reason that queries are slow. Here are some of the obvious reasons for slow queries:

- Extra heavy user load
- Hardware load
- Network load

After looking for the existence of the preceding reasons, we can pull out Management Studio and start to look at the plans of the slow queries. Most often, slow queries are apparent because either `|--Clustered Index Scan` or `|--Table Scan` shows up in the query plan, and those operations take a large percentage of time to execute. Simple, right? Essentially, it is a true enough statement that index and table scans are time consuming, but unfortunately, that really doesn't give a full picture of the process. It's hard to make specific indexing changes before knowing about usage, because the usage pattern will greatly affect these decisions, for example:

- Is a query executed once a day, once an hour, or once a minute?
- Is a background process inserting into a table rapidly? Or perhaps inserts are taking place during off hours?

Using Profiler and the Dynamic Management Views, you can watch the usage patterns of the queries that access your database, looking for slowly executing queries, poor plans, and so on. After you do this and you start to understand the usage patterns for your database, you now need to use that information to consider where to

www.it-ebooks.info

apply indexes—the final goal being that you use the information you can gather about usage and tailor an index plan to solve the overall picture.

You can't just throw around indexes to fix individual queries. Nothing comes without a price, and indexes definitely have a cost. You need to consider how indexes help and hurt the different types of operations in different ways:

- `SELECT`: Indexes can only have a beneficial effect on `SELECT` queries.
- `INSERT`: An index can only hurt the process of inserting new data into the table. As data is created in the table, there's a chance that the index will have to be modified and reorganized to accommodate the new values.
- `UPDATE`: An update requires two or three steps: find the row(s) and change the row(s), or find the row(s), delete them, and reinsert them. During the phase of finding the row, the index is beneficial, such as for a `SELECT`. Whether or not it hurts during the second phase depends on several factors, for example:
  - Did the index key value change such that it needs to be moved around to different leaf nodes?
  - Will the new value fit on an existing page, or will it require a page split? (More on that later in this section.)
- `DELETE`: The delete requires two steps: to find the row and to remove it. Indexes are beneficial to find the row, but on deletion, you might have to do some reshuffling to accommodate the deleted values from the indexes.

You should also realize that for `INSERT`, `UPDATE`, or `DELETE` operations, if triggers on the table exist (or constraints exist that execute functions that reference tables), indexes will affect those operations in the same ways as in the list. For this reason, I'm going to shy away from any generic advice about what types of columns to index. In practice, there are just too many variables to consider.

■ **Tip** Too many people index without considering the costs. Just be wary that every index you add has to be maintained. Sometimes, a query taking 1 second to execute is OK when getting it down to .1 seconds might slow down other operations considerably. The real question lies in how often each operation occurs and how much cost you are willing to suffer. The hardest part is keeping your tuning hat off until you can really get a decent profile of what operations are taking place.

For a good idea of how your current indexes and/or tables are currently being used, you can query the dynamic management view `sys.dm_db_index_usage_stats`:

```
SELECT OBJECT_NAME(i.object_id) AS object_name
      , CASE WHEN i.is_unique = 1 THEN 'UNIQUE ' ELSE '' END +
             i.TYPE_DESC AS index_type
      , i.name AS index_name
      , user_seeks, user_scans, user_lookups,user_updates
FROM sys.indexes i
        LEFT OUTER JOIN sys.dm_db_index_usage_stats AS s
```


www.it-ebooks.info

```
            ON i.object_id = s.object_id
               AND i.index_id = s.index_id
               AND database_id = db_id()
WHERE  OBJECTPROPERTY(i.object_id , 'IsUserTable') = 1
ORDER  BY 1,3;
```

This query will return the name of each object, an index type, an index name, plus the number of

- *User seeks*: The number of times the index was used in a seek operation
- *User scans*: The number of times the index was scanned in answering a query
- *User lookups*: For clustered indexes, the number of times the index was used to resolve the row locator of a nonclustered index search
- *User updates*: The number of times the index was changed by a user query

This information is very important when trying to get a feel for which indexes might need to be tuned and especially which ones are not doing their jobs, because they are mostly getting updated. You could probably boil performance tuning to a math equation if you had an advanced degree in math and a lot of time, but truthfully, it would take longer than just testing in most cases, especially if you have a good performance-testing plan for your system. Even once you know the customer's answer to these questions, you should test your database code on your performance-testing platform. Performance testing is a tricky art, but it doesn't usually take tremendous amounts of time to identify your hot spots and optimize them, and to do the inevitable tuning of queries in production.

Determining Index Usefulness

It might seem at this point that all you need to do is look at the plans of queries, look for the search arguments, and put an index on the columns, so things will improve. There's a bit of truth to this, but indexes have to be useful to be used by a query. What if the index of a 418-page book had two entries:

> General Topics 1
>
> Determining Index Usefulness 417

This means that one page was classified such that the topic started on this page, and all other pages covered general topics. This would be useless to you, unless you needed to know about indexes. One thing is for sure: you could determine that the index was useless pretty quickly. Another thing we all do with the index of a book to see if it's useful is to take a value and look it up in the index. If what you're looking for is in there (or something close), you go to the page and check it out.

SQL Server determines whether or not to use your index in much the same way. It has two specific measurements that it uses to decide if an index is useful: the *density* of values (sometimes known as the *selectivity*) and a histogram of a sample of values in the table to check against.

You can see these in detail for indexes by using `DBCC SHOW_STATISTICS`. Our table is very small, so it doesn't need stats to decide which to use. Instead, we'll look at an index in the `AdventureWorks2012` database:

```
DBCC SHOW_STATISTICS('AdventureWorks2012.Production.WorkOrder',
        'IX_WorkOrder_ProductID') WITH DENSITY_VECTOR;
DBCC SHOW_STATISTICS('AdventureWorks2012.Production.WorkOrder',
        'IX_WorkOrder_ProductID') WITH HISTOGRAM;
```


www.it-ebooks.info

This returns the following sets (truncated for space). The first tells us the size and density of the keys. The second shows the histogram of where the table was sampled to find representative values.

```
All density         Average Length      Columns
------------        --------------      ----------------------
0.004201681         4                   ProductID
1.377581E-05        8                   ProductID, WorkOrderID

RANGE_HI_KEY     RANGE_ROWS     EQ_ROWS                 DISTINCT_RANGE_ROWS    AVG_RANGE_ROWS
------------     ----------     ----------------------  -------------------    --------------
3                0              1093                    0                      1
316              0              1093                    0                      1
324              0              1093                    0                      1
327              0              1093                    0                      1
…                …              …                       …                      …
994              0              878                     0                      1
995              0              710                     0                      1
996              0              1084                    0                      1
997              0              236                     0                      1
998              0              236                     0                      1
999              0              233                     0                      1
```

I won't cover the `DBCC SHOW_STATISTICS` command in great detail, but there are a couple important things to understand. First, consider the density of each column set. The `ProductId` column is the only column that is actually declared in the index, but note that it includes the density of the index column and the clustered index key as well (it's known as the *clustering key*, which I'll cover more later in this chapter).

All the density is calculated approximately by `1 / number of distinct rows`, as shown here for the same columns as I just checked the density on:

```
--Used isnull as it is easier if the column can be null
--value you translate to should be impossible for the column
--ProductId is an identity with seed of 1 and increment of 1
--so this should be safe (unless a dba does something weird)
SELECT 1.0/COUNT(DISTINCT ISNULL(ProductID,-1)) AS density,
           COUNT(DISTINCT ISNULL(ProductID,-1)) AS distinctRowCount,
       1.0/COUNT(*) AS uniqueDensity,
           COUNT(*) AS allRowCount
FROM   AdventureWorks2012.Production.WorkOrder;
```

This returns the following:

```
density              distinctRowCount       uniqueDensity          allRowCount
--------------       ----------------       ---------------        -----------
0.004201680672       238                    0.000013775812         72591
```

You can see that the densities match. (The queries density is in a numeric type, while the DBCC is using a float, which is why they are formatted differently, but they are the same value!) The smaller the number, the better the index, and the more likely it will be easily chosen for use. There's no magic number, per se, but this value fits into the calculations of which way is best to execute the query. The actual numbers returned from this query might vary slightly from the DBCC value, as a sampled number might be used for the distinct count.

www.it-ebooks.info

The second thing to understand in the DBCC SHOW_STATISTICS output is the histogram. Even if the density of the index isn't low, SQL Server can check a given value (or set of values) in the histogram to see how many rows will likely be returned. SQL Server keeps statistics about columns in a table as well as in indexes, so it can make informed decisions as to how to employ indexes or table columns. For example, consider the following rows from the histogram (I have faked some of these results for demonstration purposes):

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
…	…	…	…	…
989	111	58	2	55.5
992	117	67	2	58.5

In the second row, the row values tell us the following:

- RANGE_HI_KEY: The sampled ProductId values were 989 and 992.
- RANGE_ROWS: There are 117 rows where the value was between 989 and 992 (noninclusive of the endpoints). These values would not be known. However, if a user used 990 as a search argument, the optimizer can now know that a maximum of 117 rows would be returned. This is one of the ways that the query plan gets the estimated number of rows for each step in a query and is one of the ways to determine if an index will be useful for an individual query.
- EQ_ROWS: There were exactly 67 rows where ProductId = 992.
- DISTINCT_RANGE_ROWS: For the row with 989, it is estimated that there are two distinct values between 989 and 992.
- AVG_RANGE_ROWS: This is the average number of duplicate values in the range, excluding the upper and lower bounds. This value is what the optimizer can expect to be the average number of rows. Note that this is calculated by RANGE_ROWS / DISTINCT_RANGE_ROWS.

One thing that having this histogram can do is allow a seemingly useless index to become valuable in some cases. For example, say you want to index a column with only two values. If the values are evenly distributed, the index would be useless. However, if there are only a few of a certain value, it could be useful (going back to the tempdb):

```
CREATE TABLE testIndex
(
    testIndex int NOT NULL IDENTITY(1,1) CONSTRAINT PKtestIndex PRIMARY KEY,
    bitValue bit NOT NULL,
    filler char(2000) NOT NULL DEFAULT (REPLICATE('A',2000)) NOT NULL
);
CREATE INDEX XtestIndex_bitValue ON testIndex(bitValue);
GO
SET NOCOUNT ON;
INSERT INTO testIndex(bitValue)
VALUES (0);
GO 50000 --runs current batch 50000 times in Management Studio.
INSERT INTO testIndex(bitValue)
VALUES (1);
GO 100 --puts 100 rows into table with value 1
```


www.it-ebooks.info

You can guess that few rows will be returned if the only value desired is 1. Check the plan for `bitValue = 0` (again using `SET SHOWPLAN ON`, or using the GUI):

```
SELECT  *
FROM    testIndex
WHERE   bitValue = 0;
```

This shows a clustered index scan:

```
|--Clustered Index Scan(OBJECT:([tempdb].[dbo].[testIndex].[PKtestIndex]),
                        WHERE:([tempdb].[dbo].[testIndex].[bitValue]=(0)))
```

However, change the 0 to a 1, and the optimizer chooses an index seek. This means that it performed a seek into the index to the first row that had a 1 as a value and worked its way through the values:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:
      ([tempdb].[dbo].[testIndex].[testIndex], [Expr1003]) WITH UNORDERED PREFETCH)
    |--Index Seek(OBJECT:([tempdb].[dbo].[testIndex].[XtestIndex_bitValue]),
                  SEEK:([tempdb].[dbo].[testIndex].[bitValue]=(1)) ORDERED FORWARD)
       |--Clustered Index Seek(OBJECT:([tempdb].[dbo].[testIndex].[PKtestIndex]),
                               SEEK:([tempdb].[dbo].[testIndex].[testIndex]=
                                     [tempdb].[dbo].[testIndex].[testIndex]) LOOKUP ORDERED
                                      FORWARD)
```

This output may look a bit odd, but this plan shows that the query processor will do the index seek to find the rows that match and then a nested loop join to the clustered index to get the rest of the data for the row (because we chose to do `SELECT *`), getting the entire data row (more on how to avoid the clustered seek in the next section).

You can see why in the histogram:

```
UPDATE STATISTICS dbo.testIndex;
DBCC SHOW_STATISTICS('dbo.testIndex', 'XtestIndex_bitValue') WITH HISTOGRAM;
```

This returns the following results (your actual values will likely vary, and in fact, in some tests, only the 0 rows showed up in the output):

```
RANGE_HI_KEY    RANGE_ROWS      EQ_ROWS         DISTINCT_RANGE_ROWS     AVG_RANGE_ROW
------------    ----------      --------        -------------------     -------------
0               0               49976.95        0                       1
1               0               123.0454        0                       1
```

The statistics gathered estimated that about 123 rows match for `bitValue = 1`. That's because statistics gathering isn't an exact science—it uses a sampling mechanism rather than checking every value (your values might vary as well). Check out the `TABLESAMPLE` clause, and you can use the same mechanisms to gather random samples of your data.

The optimizer knew that it would be advantageous to use the index when looking for `bitValue = 1`, because approximately 123 rows are returned when the index key with a value of 1 is desired, but 49,977 are returned for 0. (Your try will likely return a different value. For the rows where the bitValue was 1, I got 80 in the previous edition

www.it-ebooks.info

and 137 in a different set of tests. They are all approximately the 100 that you should expect, since we specifically created 100 rows when we loaded the table.)

This demonstration of the histogram is good, but in practice, starting in SQL Server 2008, actually building a filtered index to optimize this query may be a better practice. You might build an index such as this:

```
CREATE INDEX XtestIndex_bitValueOneOnly
       ON testIndex(bitValue) WHERE bitValue = 1;
```

The histogram for this index is definitely by far a clearer good match:

```
RANGE_HI_KEY      RANGE_ROWS          EQ_ROWS          DISTINCT_RANGE_ROWS       AVG_RANGE_ROWS
------------      ----------          --------         -------------------       --------------
1                 0                   100              0                         1
```

Whether or not the query actually uses this index will likely depend on how badly the other index would perform, which can also be dependent on hardware conditions. A histogram is, however, another tool that you can use when optimizing your SQL to see what the optimizer is using to make its choices.

■ **Tip** Whether or not the histogram includes any data where the `bitValue = 1` is largely a matter of chance. I have run this example several times, and one time, no rows were shown unless I used the `FULLSCAN` option on the `UPDATE STATISTICS` command (which isn't feasible on very large tables unless you have quite a bit of time).

As we discussed in the clustered index section, queries can be for equality or inequality. For equality searches, the query optimizer will use the single point and estimate the number of rows. For inequality, it will use the starting point and ending point of the inequality and determine the number of rows that will be returned from the query.

Indexing and Multiple Columns

So far, the indexes I've talked about were mostly on single columns, but it isn't all that often that you only need performance enhancing indexes on single columns. When multiple columns are included in the `WHERE` clause of a query on the same table, there are several possible ways you can enhance your queries:

- Having one composite index on all columns
- Creating *covering indexes* by including all columns that a query touches
- Having multiple indexes on separate columns
- Adjusting key sort order to optimize sort operations

Composite Indexes

When you include more than one column in an index, it's referred to as a *composite index*. As the number of columns grows, the effectiveness of the index is reduced for the general case. The problem is that the index is sorted by the first column values. So the second column in the index is more or less only useful if you need the first column as well. Even so, you will very often need composite indexes to optimize common queries when predicates on all of the columns involved.

www.it-ebooks.info

The order of the columns in a query is important with respect to whether a composite can and will be used. There are a couple important considerations:

- *Which column is most selective?* If one column includes unique or mostly unique values, it is possibly a good candidate for the first column. The key is that the first column is the one by which the index is sorted. Searching on the second column only is less valuable (though queries using only the second column can scan the index leaf pages for values).
- *Which column is used most often without the other columns?* One composite index can be useful to several different queries, even if only the first column of the index is all that is being used in those queries.

For example, consider this query:

```
SELECT  vegetableId, name, color, consistency
FROM    produce.vegetable
WHERE   color = 'green'
  AND   consistency = 'crunchy';
```

An index on color or consistency alone might not do the job well enough (of course, it will in this case, as `vegetable` is a very small table). If the plan that is being produced by the optimizer is to do a table scan, you might consider adding a composite index on color and consistency. This isn't a bad idea to consider, and composite indexes are great tools, but just how useful such an index will be is completely dependent on how many rows will be returned by `color = 'green'` and `consistency = 'crunchy'`.

The preceding query with existing indexes (clustered primary key on `produceId`, alternate key on `Name`, indexes on `color` and `consistency`, plus a composite index on `vegetable` and `color_`) is optimized with the following plan:

```
|--Clustered Index
        Scan(OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]),
        WHERE:([tempdb].[produce].[vegetable].[color]='green'
            AND [tempdb].[produce].[vegetable].[consistency]='crunchy'))
```

Adding an index on `color` and `consistency` seems like a good way to further optimize the query, but first, you should look at the data for these columns (consider future usage of the index too, but existing data is a good place to start):

```
SELECT color, consistency, count(*) AS [count]
FROM   produce.vegetable
GROUP BY color, consistency
ORDER BY color, consistency;
```

which returns:

```
color               consistency                 count
-------             -----------                 ------
brown               squishy                     1
green               crunchy                     1
green               leafy                       5
green               solid                       2
green               squishy                     1
orange              crunchy                     1
```


www.it-ebooks.info

```
orange              solid                    1
orange              squishy                  1
white               solid                    2
yellow              squishy                  1
```

Of course, you can't always look at all of the rows like this, so another possibility is to see which of the columns has more distinct values:

```
SELECT  COUNT(DISTINCT color) AS color,
        COUNT(DISTINCT consistency) AS consistency
FROM    produce.vegetable;
```

This query returns the following:

```
color               consistency
-----               -----------
5                   4
```

The column `consistency` has the most unique values (though not tremendously, in real practice, the difference in the number of values will often be much greater). So we add the following index:

```
CREATE INDEX Xproduce_vegetable_consistencyAndColor
     ON produce.vegetable(consistency, color);
```

The plan changes to the following (well, it would if there were lots more data in the table, so I have forced the plan to generate this particular example; otherwise, because there is such a small amount of data, the clustered index would always be used for all queries):

```
|--Nested Loops(Inner Join, OUTER REFERENCES:
        ([tempdb].[produce].[vegetable].[vegetableId]))
    |--Index Seek(OBJECT:(
        [tempdb].[produce].[vegetable].[Xproduce_vegetable_consistencyAndColor]),
        SEEK:([tempdb].[produce].[vegetable].[consistency]='crunchy'
          AND [tempdb].[produce].[vegetable].[color]='green') ORDERED FORWARD)
    |--Clustered Index Seek(OBJECT:
        ([tempdb].[produce].[vegetable].[PKproduce_vegetable]),
        SEEK:([tempdb].[produce].[vegetable].[vegetableId]=
              [tempdb].[produce].[vegetable].[vegetableId]) LOOKUP ORDERED FORWARD)
```

The execution plan does an index seek on the `Xvegetable_consistencyAndColor` index, and it uses the clustered index named `PKproduce_vegetable` to fetch the other parts of the row that were not included in the `Xvegetable_consistencyAndColor` index. (This is why the clustered index seek is noted as `LOOKUP ORDERED FORWARD`. In the graphical version of the plan, it will be called a key lookup (clustered) operator).

Keep in mind, too, exactly how the indexes will be used. If your queries mix equality comparisions and inequality comparisons, you will likely want to favor the columns you are using in equality searches first. Of course, your selectivity estimates need to be based on how selective the index will be for your situations. For example, if you are doing small ranges on very selective data in a given column, that could be the best first column in the index. If you have a question about how you think an index can help, test multiple cases and see how the plans and costs change. If you have questions about why a plan is behaving as it is, use the statistics to get more deep ideas about why an index is chosen.

www.it-ebooks.info

In the next section, I will show how you can eliminate the clustered index scan, but in general, having the scan isn't the worst thing in the world unless you are matching lots of rows. In this case, for example, the two single-row seeks would result in better performance than a full scan through the table. When the number of rows found using the nonclustered index grows large, however, a plan such as the preceding one can become very costly.

Covering Indexes

When you are only retrieving data from a table, if an index exists that has all the data values that are needed for a query, the base table needn't be touched. Back in Figure 10-10, there was a nonclustered index on the type of animal. If the name of the animal was the only data the query needed to touch, the data pages of the table wouldn't need to be accessed directly. The index *covers* all the data needed for the query and is commonly referred to as a *covering index*. The ability to create covering indexes is a nice feature, and the approach even works with clustered indexes, although with clustered indexes, SQL Server scans the lowest index structure page, because scanning the leaf nodes of the clustered index is the same as a table scan.

As a baseline to the example, let's run the following query:

```
SELECT name, color
FROM   produce.vegetable
WHERE  color = 'green';
```

The resulting plan is a simple clustered index scan:

```
|--Clustered Index Scan(OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]),
                WHERE:([tempdb].[produce].[vegetable].[color]='green'))
```

We could do a couple of things to improve the performance of the query. First, we could create a composite index on the color and name columns; it would use the color column to filter the results, and since the name column is in the index, the query processor would never need to go to the base table to get the data to return the results. If you will be filtering on the name column in some cases, that would be the best thing to do.

However, in SQL Server 2005, a new feature was added to the index-creation syntax to improve the ability to implement covering indexes—the `INCLUDE (<columns>)` clause of the `CREATE INDEX` statement. The included columns can be almost any datatype, even `(max)`-type columns. In fact, the only types that aren't allowed are `text`, `ntext`, and `image` datatypes, but you shouldn't use these types anyhow, as they're in the process of being deprecated (you should expect them eventually be completely removed from the product).

Using the `INCLUDE` keyword gives you the ability to add columns to cover a query without including those columns in the index pages, and thus without causing overhead in the use of the index. Instead, the data in the `INCLUDE` columns is added only to the leaf pages of the index. The `INCLUDE` columns won't help in index seeking, but they do eliminate the need to go to the data pages to get the data being sought.

To demonstrate, let's modify the index on vegetable color and include the `name` column:

```
DROP INDEX Xproduce_vegetable_color ON produce.vegetable;
CREATE INDEX Xproduce_vegetable_color ON produce.vegetable(color) INCLUDE (name);
```

Now, the query goes back to only touching the index, because it has all the data in the index, and this time, it doesn't even need to go to the clustered index to pick up the `name` column:

```
|--Index Seek(OBJECT:([tempdb].[produce].[vegetable].[Xproduce_vegetable_color]),
                     SEEK:([tempdb].[produce].[vegetable].[color]=[@1]) ORDERED FORWARD)
```


www.it-ebooks.info

This ability to include columns only in the leaf pages of covering indexes is incredibly useful in a lot of situations. Too many indexes with overly large keys are created to cover a query to avoid accessing the base table and end up being only good for one situation, which ends up wasting valuable resources. Now, using `INCLUDE`, you get the benefits of a covering index without the overhead of bloating the nonleaf pages of the index.

Be careful not to go crazy with covering indexes unless you can see a large benefit from them. The `INCLUDE` feature costs less to maintain than including the values in the index structure, but it doesn't make the index structure free to maintain, and it can be very costly to maintain if it references a `varchar(max)` column, as but one example. One thing you will likely notice when looking at query plans, or the missing index dynamic management views is that indexes using the `INCLUDE` feature are commonly suggested, because quite often, the key lookup is the most costly part of queries. I must include a caution about going too far and abusing covering indexes, because their use does incur a fairly heavy cost. Be careful to test that the additional overhead of duplicating data in indexes doesn't harm performance more than it helps it.

Multiple Indexes

Sometimes, we might not have a single index on a table that meets the given situation for the query optimizer to do an optimum job. In this case, SQL Server can sometimes use two or more indexes to meet the need. When processing a query with multiple indexes, SQL Server uses the indexes as if they were tables, joins them together, and returns a set of rows. The more indexes used, the larger the cost, but using multiple indexes can be dramatically faster in some cases.

Multiple indexes aren't usually something to rely on to optimize known queries. It's almost always better to support a known query with a single index. However, if you need to support ad hoc queries that cannot be foretold as a system designer, having several indexes that are useful for multiple situations might be the best idea. If you're building a read-only table, a decent starting strategy might be to index every column that might be used as a filter for a query.

My focus throughout this book has been on OLTP databases, and for that type of database, using multiple indexes in a single query isn't typical. However, it's possible that the need for using multiple indexes will arise if you have a table with several columns that you'll allow users to query against in any combination.

For example, assume you want data from four columns in a table that contains telephone listings. You might create a table for holding phone numbers called `phoneListing` with these columns: `phoneListingId`, `firstName`, `lastName`, `zipCode`, `areaCode`, `exchange`, and `number` (assuming United States–style phone numbers).

You have a clustered primary key index on `phoneListingId`, nonclustered composite indexes on `lastName` and `firstName`, one on `areaCode` and `exchange`, and another on `zipCode`. From these indexes, you can effectively perform a large variety of searches, though generally speaking, none of these will be perfect alone, but with one or two columns considered independently, it might be adequate.

For less typical names (such as Leroy Shlabotnik, for example), a person can find this name without knowing the location. For other names, hundreds and thousands of other people have the same first and last names. I always thought I was the only schmuck with the name Louis Davidson, but it turns out that there are quite a few others!

We could build a variety of indexes on these columns, such that SQL Server would only need a single index. However, not only would these indexes have a lot of columns in them but you'd need several indexes. A composite index can be useful for searches on the second and third columns, but if the first column is not included in the filtering criteria, it will require a scan, rather than a seek, of the index. Instead, for large sets, SQL Server can find the set of data that meets one index's criteria and then join it to the set of rows that matches the other index's criteria.

This technique can be useful when dealing with large sets of data, especially when users are doing ad hoc querying, and you cannot anticipate what columns they'll need until runtime. Users have to realize that they need to specify as few columns as possible, because if the multiple indexes can cover a query such as the one in the last section, the indexes will be far more likely to be used.

www.it-ebooks.info

As an example, we'll use the data already created, and add an index on the consistency column:

```
CREATE INDEX Xproduce_vegetable_consistency ON produce.vegetable(consistency);
--existing index repeated as a reminder
--CREATE INDEX Xproduce_vegetable_color ON produce.vegetable(color) INCLUDE (name);
```

We'll force the optimizer to use multiple indexes (because the sample table is far too small to require multiple indexes), so we can show how this looks in a plan:

```
SELECT  consistency, color
FROM    produce.vegetable WITH (INDEX=Xproduce_vegetable_color,
                                INDEX=Xproduce_vegetable_consistency)
WHERE   color = 'green'
 and    consistency = 'leafy';
```

This produces the following plan:

```
|--Merge Join(Inner Join, MERGE:([tempdb].[produce].[vegetable].[vegetableId])=
            ([tempdb].[produce].[vegetable].[vegetableId]),
                      RESIDUAL:([tempdb].[produce].[vegetable].[vegetableId] =
                                [tempdb].[produce].[vegetable].[vegetableId]))
      |--Index Seek(OBJECT:([tempdb].[produce].[vegetable].[Xproduce_vegetable_color]),
                     SEEK:([tempdb].[produce].[vegetable].[color]='green') ORDERED FORWARD)
      |--Index Seek(OBJECT:
                   ([tempdb].[produce].[vegetable].[Xproduce_vegetable_consistency]),
                      SEEK:([tempdb].[produce].[vegetable].[consistency]='leafy')
                                    ORDERED FORWARD)
```

Looking at a snippet of the plan for this query, you can see that there are two index seeks to find rows where `color = 'green'` and `consistency = 'leafy'`. These seeks would be fast on even a very large set, as long as the index was reasonably selective. Then, a merge join is done between the sets, because the sets can be ordered by the clustered index. (There's a clustered index on the table, so the clustering key is included in the index keys.)

Sort Order of Index Keys

While SQL Server can traverse an index in either direction (since it is a doubly linked list), sometimes sorting the keys of an index to match the sort order of some desired output can be valuable. For example, consider the case where you want to look at the hire dates of your employees, in descending order by hire date. To do that, execute the following query (in the `AdventureWorks2012` database):

```
SELECT MaritalStatus, HireDate
FROM   Adventureworks2012.HumanResources.Employee
ORDER  BY MaritalStatus ASC, HireDate DESC;
```

The plan for this query follows:

```
 |--Sort(ORDER BY:([AdventureWorks2012].[HumanResources].[Employee].[MaritalStatus] ASC,
                  [AdventureWorks2012].[HumanResources].[Employee].[HireDate] DESC))
       |--Clustered Index Scan(OBJECT:(
       [AdventureWorks2012].[HumanResources].[Employee].[PK_Employee_BusinessEntityID]))
```


www.it-ebooks.info

Next, create a typical index with the default (ascending) sort order:

```
CREATE INDEX Xemployee_maritalStatus_hireDate ON
    Adventureworks2012.HumanResources.Employee (MaritalStatus,HireDate);
```

Rechecking the plan, you will see that the plan changes to an index scan (since it can use the index to cover the query), but it still requires a sort operation.

```
|--Sort(ORDER BY:([AdventureWorks2012].[HumanResources].[Employee].[MaritalStatus] ASC,
        [AdventureWorks2012].[HumanResources].[Employee].[HireDate] DESC))
    |--Index Scan(OBJECT: ([AdventureWorks2012].[HumanResources].[Employee].
        [Xemployee_maritalStatus_hireDate]))
```

That's better but still not quite what we want. Change the index we just added to be sorted in the direction that the output is desired in:

```
DROP INDEX Xemployee_maritalStatus_hireDate ON
    Adventureworks2012.HumanResources.Employee;
GO
CREATE INDEX Xemployee_maritalStatus_hireDate ON
    AdventureWorks2012.HumanResources.Employee(MaritalStatus ASC,HireDate DESC);
```

Now, reexecute the query, and the sort is gone:

```
|--Index Scan(OBJECT:([AdventureWorks2012].[HumanResources].[Employee].
      [Xemployee_maritalStatus_hireDate]), ORDERED FORWARD)
```

In a specifically OLTP database, tweaking index sorting is not necessarily the best thing to do just to tune a single query. Doing so creates an index that will need to be maintained, which, in the end, may cost more than just paying the cost of the index scan. Creating an index in a sort order to match a query's `ORDER BY` clause is, however, another tool in your belt to enhance query performance. Consider it when an `ORDER BY` operation is done frequently enough and at a cost that is otherwise too much to bear.

Nonclustered Indexes on a Heap

Although there are rarely compelling use cases for leaving a table as a heap structure in a production OLTP database, I do want at least to show you how this works. As an example of using a nonclustered index with a heap, we'll drop the primary key on our table and replace it with a nonclustered version of the `PRIMARY KEY` constraint:

```
ALTER TABLE produce.vegetable
    DROP CONSTRAINT PKproduce_vegetable;

ALTER TABLE produce.vegetable
    ADD CONSTRAINT PKproduce_vegetable PRIMARY KEY NONCLUSTERED (vegetableID);
```

Now, we look for a single value in the table:

```
SELECT  *
FROM    produce.vegetable
WHERE   vegetableId = 4;
```


www.it-ebooks.info

The following plan will be used to execute the query:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1000]))
      |--Index Seek(
            OBJECT:([tempdb].[produce].[vegetable].[PKproduce_vegetable]),
              SEEK:([tempdb].[produce].[vegetable].[vegetableId]=
                    CONVERT_IMPLICIT(int,[@1],0)) ORDERED FORWARD)
      |--RID Lookup(OBJECT:([tempdb].[produce].[vegetable]),
              SEEK:([Bmk1000]=[Bmk1000]) LOOKUP ORDERED FORWARD)
```

First, we probe the index for the value; then, we have to look up the row from the row ID (RID) in the index (the RID lookup operator). The most important thing I wanted to show in this section was the RID lookup operator, so you can identify this on a plan and understand what is going on. This RID lookup is the same as the clustered index seek or row lookup operator. However, instead of using the clustering key, it uses the physical location of the row in the table. (As discussed earlier in this chapter, keeping this physical pointer stable is why the heap structure uses forwarding pointers instead of page splits and why it is generally considered best practice to have every table be a clustered table.)

Using Unique Indexes

An important index setting is `UNIQUE`. In the design of the tables, `UNIQUE` and `PRIMARY KEY` constraints were created to enforce keys. Behind the scenes, SQL Server employs unique indexes to enforce uniqueness over a column or group of columns. SQL Server uses them for this purpose because to determine if a value is unique, you have to look it up in the table. Because SQL Server uses indexes to speed access to the data, you have the perfect match.

Enforcing uniqueness is a business rule, and as I covered in Chapter 6, the rule of thumb is to use `UNIQUE` or `PRIMARY` constraints to enforce uniqueness on a set of columns. Now, as you're improving performance, use unique indexes when the data you're indexing allows it.

For example, say you're building an index that happens to include a column (or columns) that is already a part of another unique index. Another possibility might be if you're indexing a column that's naturally unique, such as a GUID. It's up to the designer to decide if this GUID is a key or not, and that depends completely on what it's used for. Using unique indexes lets the optimizer determine more easily the number of rows it has to deal with in an equality operation.

Also note that it's important for the performance of your systems that you use unique indexes whenever possible, as they enhance the SQL Server optimizer's chances of predicting how many rows will be returned from a query that uses the index. If the index is unique, the maximum number of rows that can be returned from a query that requires equality is one. This is common when working with joins.

Index Usage Scenarios

So far, we've dealt with the mechanics of indexes and basic situations where they're useful. Now, we need to talk about a few special uses of indexes that deserve some preplanning or understanding to use.

I'll discuss the following topics:

- Indexing foreign keys
- Indexing views to optimize denormalization

www.it-ebooks.info

Indexing Foreign Keys

Foreign key columns are a special case where we often need an index of some type. This is because we build foreign keys so we can match up rows in one table to rows in another. For this, we have to take a value in one table and match it to another.

In an OLTP database that has proper constraints on alternate keys, often, we won't need to index foreign keys beyond what we're given with the unique indexes that are built as part of the structure of the database. This is probably why SQL Server doesn't implement indexes for us when creating a foreign key constraint.

However, it's important to make sure that any time you have a foreign key constraint declared, there's the potential for need of an index whenever you have a parent table and you want to see the children of the row. A special and important case where this type of access is essential is when you have to delete the parent row in any relationship, even one of a domain type that has a very low cardinality. I will usually apply an index to all foreign keys as a default in my model, removing the index if it is obvious that it is not helping performance. One of the queries that will be presented in the Index Dynamic Management Views section will identify how or if indexes are being used.

Say you have five values in the parent table and five million in the child. For example, consider the case of a click log for a sales database, a snippet of which is shown in Figure 10-15.

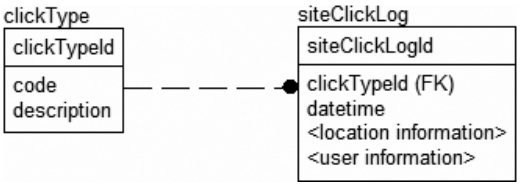


Figure 10-15. *Sample foreign key relationship*

Consider that you want to delete a `clickType` row that someone added inadvertently. Creating the row took several milliseconds. Deleting it shouldn't take long at all, right? Well, even if there isn't a single value in the table, if you don't have an index on the foreign key in the `siteClickLog` table, it will take just over 10 seconds longer than eternity. Even though the value doesn't exist in the table, the query processor would need to touch and check all five million rows for the value. From the statistics on the column, it can guess how many rows might exist, but it can't definitively know that there is or is not a value because statistics are maintained asyncronously. (Ironically, because it is an existence search, it could fail on the first row it checked, but to successfully delete the row, every child row must be touched.)

However, if you have an index, deleting the row (or knowing that you can't delete it) will take a very short period of time, because in the upper pages of the index, you'll have all the unique values in the index, in this case, five values. There will be a fairly substantial set of leaf pages for the index, but only one page in each index level, usually no more than three or four pages, will need to be touched before the query processor can determine the existence of a row out of millions of rows. When `NO ACTION` is specified for the relationship, if just one row is found, the operation could be stopped. If you have cascading operations enabled for the relationship, the cascading options will need the index to find the rows to cascade to.

www.it-ebooks.info

This adds more decisions when building indexes. Is the cost of building and maintaining the index during creation of millions of `siteClickLog` rows justified, or do you just bite the bullet and do deletes during off hours? Add a trigger such as the following, ignoring error handling in this example for brevity:

```
CREATE TRIGGER clickType$insteadOfDelete
ON clickType
INSTEAD OF DELETE
AS
    INSERT  INTO clickType_deleteQueue (clickTypeId)
    SELECT  clickTypeId
    FROM    inserted;
```

Then, you let your queries that return lists of `clickType` rows check this table when presenting rows to the users:

```
SELECT  code, description, clickTypeId
FROM    clickType
WHERE   NOT EXISTS (SELECT *
                    FROM  clickType_deleteQueue
                    WHERE clickType.clickTypeId =
                              clickType_deleteQueue.clickTypeId);
```

Now, assuming all code follows this pattern, the users will never see the value, so it won't be an issue (at least not in terms of values being used; performance will suffer obviously). Then, you can delete the row in the wee hours of the morning without building the index. Whether or not an index proves useful generally depends on the purpose of the foreign key. I'll mention specific types of foreign keys individually, each with their own signature usage:

- *Domain tables*: Used to implement a defined set of values and their descriptions
- *Ownership*: Used to implement a multivalued attribute of the parent
- *Many-to-many resolution*: Used to implement a many-to-many relationship physically
- *One-to-one relationships*: Cases where a parent may have only a single value in the related table

We'll look at examples of these types and discuss when it's appropriate to index them before the typical trial-and-error performance tuning, where the rule of thumb is to add indexes to make queries faster, while not slowing down other operations that create data.

In all cases, deleting the parent row requires a table scan of the child if there's no index on the child row. This is an important consideration if there are deletes.

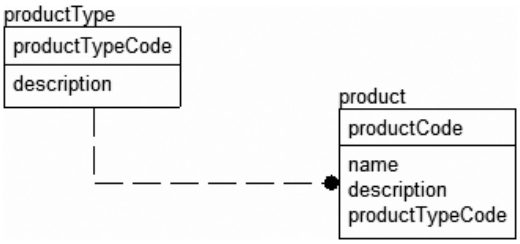


Figure 10-16. *Sample domain table relationship*

www.it-ebooks.info

Domain Tables

You use a domain table to enforce a domain using a table, rather than using a scalar value with a constraint. This is often done to enable a greater level of data about the domain value, such as a descriptive value. For example, consider the tables in Figure 10-16.

In this case, there are a small number of rows in the `productType` table. It's unlikely that an index on the `product.productTypeCode` column would be of any value in a join, because you'll generally be getting a `productType` row for every row you fetch from the `product` table.

What about the other direction, when you want to find all products of a single type? This can be useful if there aren't many products, but in general, domain type tables don't have enough unique values to merit an

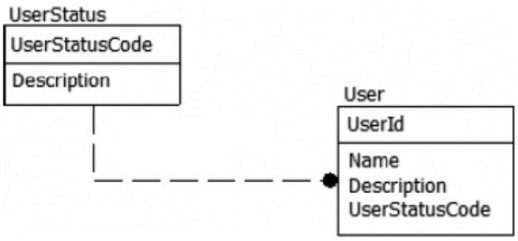


Figure 10-17. *Sample domain table relationship with low cardinality*

index. The general advice is that tables of this sort don't need an index on the foreign key values, by default. Of course, deleting `productType` rows would need to scan the entire `productType`.

On the other hand, as discussed in the section introduction, sometimes an index can be useful when there are limited numbers of some value. For example, consider a `user` to `userStatus` relationship illustrated in Figure 10-17.

In this case, most users would be in the database with an active status. However, when a user was deactivated, you might need to do some action for that user. Since the number of inactive users would be far fewer than active users, it might be useful to have an index (possibly a filtered index) on the `userStatusCode` column for that purpose.

Ownership Relationships

Some tables have no meaning without the existence of another table and pretty much exist to as part of another table (due to the way relational design works). When I am thinking about an ownership relationship, I am

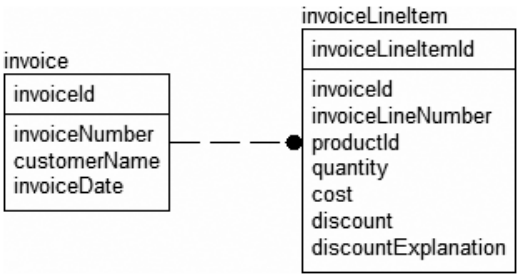


Figure 10-18. *Sample ownership relationship*

www.it-ebooks.info

thinking of relationships that implement multivalued attributes for a row, just like an array in a procedural language does for an object. The main performance characteristic of this situation is that most of the time when the parent row is retrieved, the child rows are retrieved as well. You'll be less likely to need to retrieve a child row and then look for the parent row.

For example, take the case of an invoice and its line items in Figure 10-18.

In this case, it's essential to have an index on the `invoiceLineItem.invoiceId` column. Most access to the `invoiceLineItem` table results from a user's need to get an invoice first. This situation is also ideal for an index because, usually, it will turn out to be a very selective index (unless you have large numbers of items and few sales).

Note that you should already have a `UNIQUE` constraint (and a unique index as a consequence of this) on the alternate key for the table—in this case, `invoiceId` and `invoiceLineNumber`. Therefore, you probably wouldn't need to have an index on just `invoiceId`. What might be in question would be whether or not the index on `invoiceId` and `invoiceLineNumber` ought to be clustered. If you do most of your select operations using the `invoiceId`, this actually can be a good idea. However, you should be careful in this case because you can actually do a lot more fetches on the primary key value, since update and delete operations start out performing like a `SELECT` before the modification. For example, the application may end up doing one query to get the invoice line items and then update each row individually to do some operation. So always watch the activity in your database and tune accordingly.

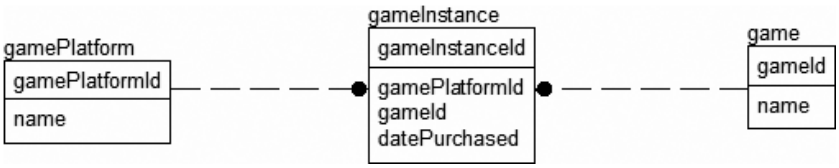


Figure 10-19. *Sample many-to-many relationship*

Many-to-Many Resolution Table Relationships

When we have a many-to-many relationship, there certainly needs to be an index on the two migrated keys from the two parent tables. Think back to our Chapter 7 of games owned on a given platform (diagram repeated in Figure 10-19).

In this case, you should already have a unique index on `gamePlatformId` and `gameId`, and one of the two will necessarily be first in the composite index. If you need to search for both keys independently of one another, you may want to create an index on each column individually (or at least the column that is listed second in the uniqueness constraint's index).

Take this example. If we usually look up a game by name (which would be alternate key indexed) and then get the platforms for this game, an index only on `gameInstance.gameId` would be much more useful and two-thirds the size of the alternate key index (assuming a clustering key of `gameInstanceId`).

www.it-ebooks.info

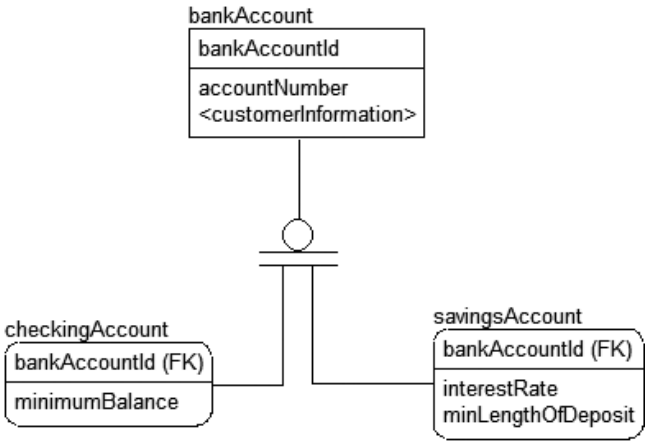


Figure 10-20. *Sample one-to-one relationship*

One-to-One Relationships

One-to-one relationships generally require some form of unique index on the key in the parent table as well as on the migrated key in the child table. For example, consider the subclass example of a `bankAccount`, shown in Figure 10-20.

In this case, because these are one-to-one relationships, and there are already indexes on the primary key of each table, no other indexes would need to be added to effectively implement the relationship.

Indexing Views

I mentioned the use of persisted calculated columns in Chapter 6 for optimizing denormalizations for a single row, but sometimes, your denormalizations need to span multiple rows and include things like summarizations. In this section, I will introduce a way to take denormalization to the next level, using *indexed views*.

Indexing a view basically takes the virtual structure of the view and makes it a physical entity, albeit one managed completely by the query processor. The data to resolve queries with the view is generated as data is modified in the table, so access to the results from the view is just as fast as if it were an actual table. Indexed views give you the ability to build summary tables without any kind of manual operation or trigger; SQL Server automatically maintains the summary data for you. Creating indexed views is as easy as writing a query.

The benefits are twofold when using indexed views. First, when you use an indexed view directly in any edition of SQL Server, it does not have to do any calculations (editions less than Enterprise must specify `NOEXPAND` as a hint). Second, in the Enterprise Edition or greater (plus the Developer Edition), SQL Server automatically considers the use of an indexed view whenever you execute any query, even if the query doesn't reference the view but the code you execute uses the same aggregates. SQL Server accomplishes this index view assumption by matching the executed query to each indexed view to see whether that view already has the answer to something you are asking for.

For example, we could create the following view on the product and sales tables in the `Adventureworks2012` database. Note that only schema-bound views can be indexed as this makes certain that the tables and structures

www.it-ebooks.info

that the index is created upon won't change underneath the view. (A more complete list of requirements is presented after the example.). Note that you actually have to be in the AdventureWorks2012 database, unlike other examples where I have addressed items to the database, because index views have strict requirements, which I will cover in more detail in the next section of this chapter.

```
USE AdventureWorks2012;
GO
CREATE VIEW Production.ProductAverageSales
WITH SCHEMABINDING
AS
SELECT  Product.ProductNumber,
        SUM(SalesOrderDetail.LineTotal) AS TotalSales,
        COUNT_BIG(*) AS CountSales --must use COUNT_BIG for indexed view
FROM    Production.Product as Product
            JOIN Sales.SalesOrderDetail AS SalesOrderDetail
                ON Product.ProductID=SalesOrderDetail.ProductID
GROUP   BY Product.ProductNumber;
```

This would do the calculations at execution time. We run the following query:

```
SELECT  ProductNumber, TotalSales, CountSales
FROM    Production.ProductAverageSales;
```

The plan looks like this:

```
|--Hash Match(Inner Join, HASH:([SalesOrderDetail].[ProductID])= ([Product].[ProductID]))
      |--Hash Match(Aggregate, HASH:([SalesOrderDetail].[ProductID])
                            DEFINE:([Expr1004]=SUM([AdventureWorks2012].[Sales].
                                    [SalesOrderDetail].[LineTotal] as
                                    [SalesOrderDetail].[LineTotal]), [Expr1005]=COUNT(*)))
      | |--Compute Scalar(DEFINE:([SalesOrderDetail].[LineTotal]=
             [AdventureWorks2012].[Sales].[SalesOrderDetail].[LineTotal] as
                 [SalesOrderDetail].[LineTotal]))
      | |--Compute Scalar(DEFINE:([SalesOrderDetail].[LineTotal]=
                                                  isnull((CONVERT_IMPLICIT(numeric(19,4),
             [AdventureWorks2012].[Sales].[SalesOrderDetail].[UnitPrice] as
                 [SalesOrderDetail].[UnitPrice],0)*
                 ((1.0)-CONVERT_IMPLICIT(numeric(19,4),
             [AdventureWorks2012].[Sales].[SalesOrderDetail].
                                    [UnitPriceDiscount] as [SalesOrderDetail].
                                    [UnitPriceDiscount],0)))* CONVERT_IMPLICIT(numeric(5,0),
             [AdventureWorks2012].[Sales].[SalesOrderDetail].[OrderQty]
                   as [SalesOrderDetail].[OrderQty],0),(0.000000))))
      | |--Clustered Index
             Scan(OBJECT:([AdventureWorks2012].[Sales].[SalesOrderDetail].
             [PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID]
                       AS [SalesOrderDetail]))
      |--Index Scan(OBJECT:(
             [AdventureWorks2012].[Production].[Product].AK_Product_ProductNumber] AS [Product]))
```


www.it-ebooks.info

This is a big plan for such a small query, for sure, and it's hard to follow. It scans the `SalesOrderDetail` table, computes our scalar values, and then does a hash match aggregate and a hash match join to join the two sets together. For further reading on the join types, again consider a book in Kalen Delaney's *SQL Server Internals* series.

This query executes pretty fast on my 1.8-GHz 8-GB laptop, but there's a slight but noticeable delay. Say this query wasn't fast enough, or it used too many resources to execute, and it is used extremely often. In this case, we might add an index on the view. Note that it is a clustered index, as the data pages will be ordered based on the key we chose. Consider your structure of the index just like you would on a physical table.

```
CREATE UNIQUE CLUSTERED INDEX XPKProductAverageSales ON
                        Production.ProductAverageSales(ProductNumber);
```

SQL Server would then materialize the view and store it. Now, our queries to the view will be *very* fast. However, although we've avoided all the coding issues involved with storing summary data, we have to keep our data up to date. Every time data changes in the underlying tables, the index on the view changes its data, so there's a performance hit in maintaining the index for the view. Hence, indexing views means that performance is great for reading but not necessarily for updating.

Now, run the query again:

```
SELECT  ProductNumber, TotalSales, CountSales
FROM    Production.ProductAverageSales;
```

The plan looks like the following:

```
|--Clustered Index Scan(OBJECT:(
      [AdventureWorks2012].[Production].[ProductAverageSales].[XPKProductAverageSales]))
```

No big deal, right? We expected this result because we directly queried the view. On my test system, running the Developer Edition (which is functionally comparable to the Enterprise Edition), you get a great insight into how cool this feature is in the following query for getting the average sales per product:

```
SELECT  Product.ProductNumber, SUM(SalesOrderDetail.LineTotal) / COUNT(*)
FROM    Production.Product AS Product
            JOIN Sales.SalesOrderDetail AS SalesOrderDetail
                  ON Product.ProductID=SalesOrderDetail.ProductID
GROUP BY Product.ProductNumber;
```

We'd expect the plan for this query to be the same as the first query of the view was, because we haven't referenced anything other than the base tables, right? I already told you the answer, so here's the plan:

```
| -- Compute
Scalar(DEFINE:(
        [Expr1006]=[AdventureWorks2012].[Production].ProductAverageSales].
        [totalSales]/CONVERT_IMPLICIT(numeric(10,0),[Expr1005],0)))
    |--Compute Scalar(DEFINE:([Expr1005]=
            CONVERT_IMPLICIT(int,[AdventureWorks2012].[Production].
            [ProductAverageSales].[averageTotal],0)))
      |--Clustered Index
             Scan(OBJECT:([AdventureWorks2012].[Production].[ProductAverageSales].
                         [XPKProductAverageSales]))
```


www.it-ebooks.info

There are two scalar computes—one for the division and one to convert the `bigint` from the `COUNT_BIG(*)` to an integer—and the other to scan through the indexed view's clustered index. You will notice that the plan references the `ProductAverageSales` indexed view, and we did not reference it directly in our query. The ability to use the optimizations from an indexed view indirectly is a neat feature that allows you to build in some guesses as to what ad hoc users will be doing with the data and giving them performance they didn't even ask for.

■ **Tip** The indexed view feature in the Enterprise Edition and greater can also come in handy for tuning third-party systems that work on an API that is not tunable in a direct manner (that is, to change the text of a query to make it more efficient).

There are some pretty heavy caveats, though. The restrictions on what can be used in a view, prior to it being indexed, are fairly tight. The most important things that *cannot* be done are as follows:

- Use the `SELECT *` syntax—columns must be explicitly named.
- Use a CLR user defined aggregate.
- Use `UNION`, `EXCEPT`, or `INTERSECT` in the view.
- Use any subqueries.
- Use any outer joins or recursively join back to the same table.
- Specify `TOP` in the `SELECT` clause.
- Use `DISTINCT`.
- Include a `SUM()` function if it references more than one column.
- Use `COUNT(*)`, though `COUNT_BIG(*)` is allowed.
- Use almost any aggregate function against a nullable expression.
- Reference any other views, or use CTEs or derived tables.
- Reference any nondeterministic functions.
- Reference data outside the database.
- Reference tables owned by a different owner.

And this isn't all. You must meet several pages of requirements, documented in SQL Server Books Online in the section "Creating Indexed Views." However, these are the most significant ones that you need to consider before using indexed views.

Although this might all seem pretty restrictive, there are good reasons for all these rules. Maintaining the indexed view is analogous to writing our own denormalized data maintenance functions. Simply put, the more complex the query to build the denormalized data, the greater the complexity in maintaining it. Adding one row to the base table might cause the view to need to recalculate, touching thousands of rows.

Indexed views are particularly useful when you have a view that's costly to run, but the data on which it's based doesn't change a tremendous amount. As an example, consider a decision-support system where you load data once a day. There's overhead either maintaining the index, or possibly just rebuilding it, but if you can build the index during off hours, you can omit the cost of redoing joins and calculations for every view usage.

www.it-ebooks.info

■ **Tip** With all the caveats, indexed views can prove useless for some circumstances. An alternative method is to materialize the results of the data by inserting the data into a permanent table. For example, for our sample query, we'd create a table with three columns (`productNumber`, `totalSales`, and `countSales`), and then we'd do an `INSERT INTO ProductAverageSales SELECT . . .`. We'd put the results of the query in this table. Any query works here, not just one that meets the strict guidelines. It doesn't help out users making ad hoc queries who don't directly query the data in the table, but it certainly improves performance of queries that directly access the data, particularly if perfect results are not needed, since data will usually be a little bit old due to the time required to refresh the data.

Index Dynamic Management View Queries

In this section, I want to provide a couple of queries that use the dynamic management views that you may find handy when you are tuning your system or index usage. In SQL Server 2005, Microsoft added a set of objects (views and table-valued functions) to SQL Server that gave us access to some of the deep metadata about the performance of the system. A great many of these objects are useful for managing and tuning SQL Server, and I would suggest you do some reading about these objects (not to be overly self serving, but the book *Performance Tuning with SQL Server Dynamic Management Views (High Performance SQL Server)* by Tim Ford and myself published by Simple-Talk is *my* favorite book on the subject!). I do want to provide you with some queries that will likely be quite useful for you when doing any tuning using indexes.

Missing Indexes

The first query we'll discuss will provide you a peek at what indexes the optimizer thought might be useful. It can be very helpful when tuning a database, particularly a very busy databsase executing thousands of queries a minute. I have personally used it to tune third-party systems where I didn't have a lot of access to the queries in the system and using profiler was simply far too cumbersome with too many queries to effectively tune manually.

It uses three of the dynamic management views that are part of the missing index family of objects. These are:

- `sys.dm_db_missing_index_groups`: This relates missing index groups with the indexes in the group. There is only one index to a group, as of the 2011 release.
- `sys.dm_db_missing_index_group_stats`: This provides statistics of how much the indexes in the group (only one in 2005; see the previous section) would have helped (and hurt) the system.
- `sys.dm_db_missing_index_details`: This provides information about the index that the optimizer would have chosen to have available.

The query is as follows. I won't attempt to build a scenario where you can test this functionality out in this book, but run the query on one of your development servers and check the results. The results will likely make you want to run it on your production server:

```
SELECT  ddmid.statement AS object_name, ddmid.equality_columns, ddmid.inequality_columns,
        ddmid.included_columns, ddmigs.user_seeks, ddmigs.user_scans,
        ddmigs.last_user_seek, ddmigs.last_user_scan, ddmigs.avg_total_user_cost,
        ddmigs.avg_user_impact, ddmigs.unique_compiles
FROM    sys.dm_db_missing_index_groups AS ddmig
```


www.it-ebooks.info

```
            JOIN sys.dm_db_missing_index_group_stats AS ddmigs
                  ON ddmig.index_group_handle = ddmigs.group_handle
            JOIN sys.dm_db_missing_index_details AS ddmid
                  ON ddmid.index_handle = ddmig.index_handle
ORDER BY ((user_seeks + user_scans) * avg_total_user_cost * (avg_user_impact * 0.01)) DESC;
```

The query returns the following information about the structure of the index that might have been useful:

- `object_name`: This is the database and schema qualified object name of the object that the index would have been useful on. The data returned is for all databases on the entire server
- `equality_columns`: These are the columns that would have been useful, based on an equality predicate. The columns are returned in a comma-delimited list.
- `inequality_columns`: These are the columns that would have been useful, based on an inequality predicate (which as we have discussed is any comparison other than `column = value` or `column in (value, value1)`).
- `included_columns`: These columns, if added to the index via an `INCLUDE` clause, would have been useful to cover the query results and avoid a key lookup operation in the plan.

As discussed earlier, the equality columns would generally go first in the index column definition, but it isn't guaranteed that that will make the correct index. These are just guidelines, and using the next DMV query I will present, you can discover if the index you create turns out to be of any value.

- `unique_compiles`: The number of plans that have been compiled that might have used the index
- `user_seeks`: The number of seek operations in user queries might have used the index
- `user_scans`: The number of scan operations in user queries that might have used the index
- `last_user_seek`: The last time that a seek operation might have used the index
- `last_user_scan`: The last time that a scan operation might have used the index
- `avg_total_user_cost`: Average cost of queries that could have been helped by the group of indexes
- `avg_user_impact`: The percentage of change in cost that the index is estimated to make for user queries

Note that I sorted the query results as `(user_seeks + user_scans) * avg_total_user_cost * (avg_user_impact * 0.01)` based on the initial blog I read using the missing indexes, "Fun for the Day" – Automated Auto-Indexing (http://blogs.msdn.com/queryoptteam/archive/2006/06/01/613516.aspx). I generally use some variant of that to determine what is most important. For example, I might use `order by (user_seeks + user_scans)` to see what would have been useful the most times. It really just depends on what I am trying to scan; with all such queries, it pays to try out the query and see what works for your situation.

To use the output, you can create a `CREATE INDEX` statement from the values in the four structural columns. Say you received the following results:

www.it-ebooks.info

```
object_name                          equality_columns          inequality_columns
-------------------------------      ------------------------  ------------------
databasename.schemaname.tablename    columnfirst, columnsecond columnthird

included_columns
------------------------
columnfourth, columnfififth
```

You could build the following index to satisfy the need:

```
CREATE INDEX XName ON databaseName.schemaName.TableName(columnfirst, columnsecond, columnthird)
INCLUDE (columnfourth, columnfififth);
```

Next, see if it helps out performance in the way you believed it might. And even if you aren't sure of how the index might be useful, create it and just see if it has an impact.

Books Online lists the following limitations to consider:

- It is not intended to fine-tune an indexing configuration.
- It cannot gather statistics for more than 500 missing index groups.
- It does not specify an order for columns to be used in an index.
- For queries involving only inequality predicates, it returns less accurate cost information.
- It reports only include columns for some queries, so index key columns must be manually selected.
- It returns only raw information about columns on which indexes might be missing. This means the information returned may not be sufficient by itself without additional processing before building the index.
- It does not suggest filtered indexes.
- It can return different costs for the same missing index group that appears multiple times in XML showplans.
- It does not consider trivial query plans.

Probably the biggest concern is that it can specify a lot of overlapping indexes, particularly when it comes to included columns, since each entry could have been specifically created for distinct queries. For very busy systems, you may find a lot of the suggestions include very large sets of include columns that you may not want to implement.

However, limitations aside, the missing index dynamic management view are amazingly useful to help you see places where the optimizer would have liked to have an index and one didn't exist. This can greatly help diagnose very complex performance/indexing concerns, particularly ones that need a large amount of `INCLUDE` columns to cover complex queries. This feature is turned on by default and can only be disabled by starting SQL Server with a command line parameter of `-x`. This will, however, disable keeping several other statistics like CPU time and cache-hit ratio stats.

Using this feature and the query in the next section that can tell you what indexes have been used, you can use these index suggestions in an experimental fashion, just building a few of the indexes and see if they are used and what impact they have on your performance tuning efforts.

www.it-ebooks.info

Index Utilization Statistics

The second query gives statistics on how an index has been used to resolve queries. Most importantly, it tells you the number of times a query was used to find a single row (`user_seeks`), a range of values, or to resolve a non-unique query (`user_scans`), if it has been used to resolve a bookmark lookup (`user_lookups`), and how many changes to the index (`user_updates`). If you want deeper information on how the index was modified, check `sys.dm_db_index_operational_stats`). The query makes use of the `sys.dm_db_index_usage_stats` object that provides just what the names says, usage statistics:

```
SELECT  OBJECT_SCHEMA_NAME(indexes.object_id) + '.' +
        OBJECT_NAME(indexes.object_id) AS objectName,
        indexes.name,
        CASE WHEN is_unique = 1 THEN 'UNIQUE '
            else '' END + indexes.type_desc AS index_type,
        ddius.user_seeks, ddius.user_scans, ddius.user_lookups,
        ddius.user_updates, last_user_lookup, last_user_scan, last_user_seek,last_user_update
FROM    sys.indexes
           LEFT OUTER JOIN sys.dm_db_index_usage_stats AS ddius
               ON indexes.object_id = ddius.object_id
                   AND indexes.index_id = ddius.index_id
                   AND ddius.database_id = DB_ID()
ORDER BY ddius.user_seeks + ddius.user_scans + ddius.user_lookups DESC;
```

The query (as written) is database dependent in order to look up the name of the index in `sys.indexes`, which is a database-level catalog view. The `sys.dm_db_index_usage_stats` object returns all indexes (including heaps and the clustered index) from the entire server (there will not be a row for `sys.dm_db_index_usage_stats` unless the index has been used since the last time the server has been started). The query will return all indexes for the current database (since the DMV is filtered on `DB_ID()` in the join criteria) and will return:

- `object_name`: Schema-qualified name of the table.
- `index_name`: The name of the index (or table) from `sys.indexes`.
- `index_type`: The type of index, including uniqueness and clustered/nonclustered.
- `user_seeks`: The number of times the index has been used in a user query in a seek operation (one specific row).
- `user_scans`: The number of times the index has been used by scanning the leaf pages of the index for data.
- `user_lookups`: For clustered indexes only, this is the number of times the index has been used in a bookmark lookup to fetch the full row. This is because nonclustered indexes use the clustered indexes key as the pointer to the base row.
- `user_updates`: The number of times the index has been modified due to a change in the table's data.
- `last_user_seek`: The date and time of the last user seek operation.
- `last_user_scan`: The date and time of the last user scan operation.
- `last_user_lookup`: The date and time of the last user lookup operation.
- `last_user_update`: The date and time of the last user update operation.

www.it-ebooks.info

There are also columns for system utilizations of the index in operations such as automatic statistics operations: system_seeks, system_scans, system_lookups, system_updates, last_system_seek, last_system_scan, last_system_lookup, and last_system_update.

This is one of the most interesting views that I often use in performance tuning. It gives you the ability to tell when indexes are *not* being used. It is easy to see when an index is being used by a query by simply looking at the plan. But now, using this dynamic management view, you can see over time what indexes are used, not used, and probably more importantly, updated many, many times without ever being used.

Fragmentation

One of the biggest tasks for the DBA of a system is to make sure that the structures of indexes and tables are within a reasonable tolerance. You can decide whether to reorganize or to rebuild using the criteria stated by SQL Server Books Online in the topic for the dynamic management view sys.dm_db_index_physical_stats. You can check the FragPercent column and REBUILD indexes with greater than 30% fragmentation and REORGANIZE those that are just lightly fragmented.

```
SELECT  s.[name] AS SchemaName,
        o.[name] AS TableName,
        i.[name] AS IndexName,
        f.[avg_fragmentation_in_percent] AS FragPercent,
        f.fragment_count ,
        f.forwarded_record_count --heap only
FROM    sys.dm_db_index_physical_stats(DB_ID(), NULL, NULL, NULL, DEFAULT) f
        JOIN sys.indexes AS i
            ON f.[object_id] = i.[object_id] AND f.[index_id] = i.[index_id]
        JOIN sys.objects AS o
            ON i.[object_id] = o.[object_id]
        JOIN sys.schemas AS s
            ON o.[schema_id] = s.[schema_id]
WHERE o.[is_ms_shipped] = 0
  AND i.[is_disabled] = 0; -- skip disabled indexes
```

sys.dm_db_index_physical_stats will give you a lot more information about the internal physical structures of your tables and indexes than I am making use of here. If you find you are having a lot of fragmentation, adjusting the fill factor of your tables or indexes (specified as a percentage of page size to leave empty for new rows) in CREATE INDEX and PRIMARY KEY and UNIQUEconstraint CREATE/ALTER DDL statements can help tremendously. How much space to leave will largely depend on your exact situation, but minimally, you want to leave approximately enough space for one full additional row to be added to each page.

Best Practices

Indexing is a complex subject, and even though this is not a short chapter, we've only scratched the surface. The following best practices are what I use as a rule of thumb when creating a new database solution. Note that I assume that you've applied UNIQUE constraints in all places where they make logical sense. These constraints most likely should be there, even if they slow down your application (there are exceptions, but if a set of values needs to be unique, it needs to be unique). From there, it's all a big tradeoff. The first rule is the most important.

www.it-ebooks.info

- *There are few reasons to add indexes to tables without testing*: Add nonconstraint indexes to your tables only as needed to enhance performance. In many cases, it will turn out that no index is needed to achieve decent performance. A caveat can be foreign key indexes.
- *Choose clustered index keys wisely*: All nonclustered indexes will use the clustering key as their row locator, so the performance of the clustered index will affect all other index utilization. If the clustered index is not extremely useful, it can affect the other indexes as well.
- *Keep indexes as thin as possible*: Only index the columns that are selective enough in the main part of the index. Use the `INCLUDE` clause on the `CREATE INDEX` statement if you want to include columns only to cover the data used by a query.
- *Consider several thin indexes rather than one monolithic index*: SQL Server can use multiple indexes in a query efficiently. This can be a good tool to support ad hoc access where the users can choose between multiple situations.
- *Be careful of the cost of adding an index*: When you insert, update, or delete rows from a table with an index, there's a definite cost to maintaining the index. New data added might require page splits, and inserts, updates, and deletes can cause a reshuffling of the index pages.
- *Carefully consider foreign key indexes*: If child rows are selected because of a parent row (including on a foreign key checking for children on a delete operation), an index on the columns in a foreign key is generally a good idea.
- `UNIQUE` *constraints are used to enforce uniqueness, not unique indexes*: Unique indexes are used to enhance performance by telling the optimizer that an index will only return one row in equality comparisons. Users shouldn't get error messages from a unique *index* violation.
- *Experiment with indexes to find the combination that gives the most benefit*: Using the missing index and index usage statistics dynamic management views, you can see what indexes the optimizer needed or try your own, and then see if your choices were ever used by the queries that have been executed.

Apply indexes during your design in a targeted fashion, making sure not to overdesign for performance too early in the process. The normalization pattern is built to give great performance as long as you design for the needs of the users of the system, rather than in an academic manner taking things to the extreme that no one will ever use. The steps we have covered through this book for proper indexing are:

- Apply all the `UNIQUE` constraints that need to be added to enforce necessary uniqueness for the integrity of the data (even if the indexes are never used for performance, though generally they will).
- Minimally, index all foreign key constraints where the parent table is likely to be the driving force behind fetching rows in the child (such as invoice to invoice line item).
- Start performance testing, running load tests to see how things perform.
- Identify queries that are slow, and consider the following:
  - Add indexes using any tools you have at your disposal.
  - Eliminate clustered index row lookups by covering queries, possibly using the `INCLUDE` keyword on indexes.

www.it-ebooks.info

- Materialize query results, either by indexed view or by putting results into permanent tables.
- Work on data location strategies with filegroups, partitioning, and so on.
- Consider compression if you are using Enterprise or Data Center edition to lower the amount of data that is stored on disk.

Summary

In the first nine chapters of this book, we worked largely as if the relational engine was magical like the hat that brought Frosty to life and that the engine could do almost anything as long as we followed the basic relational principals. Magic, however, is almost always an illusion facilitated by the hard work of someone trying to let you see only what you want to see. In this chapter, we left the world of relational programming and took a peak under the covers to see what makes the magic work, which turns out to be lots and lots of code that has been evolving for the past 16 plus years (just counting from the major rewrite of SQL Server in version 7.0.) Much of the T-SQL code written for version 1.0 would work today, and almost all of it would run with a little bit of translation, yet probably none of the engine code from 1.0 persists today.

This is one of the reasons why, in this book on design, my goals for this chapter are not to make you an expert on the internals of SQL Server but rather to give you an overview of how SQL Server works enough to help guide your designs and understand the basics of performance.

We looked at the physical structure of how SQL Server stores data, which is separate from the database-schema-table-column model that is natural in the relational model. Physically speaking, for normal row data, database files are the base container. Files are grouped into filegroups, and filegroups are owned by databases. You can get some control over SQL Server I/O by where you place the files.

Inside files, data is managed in extents of 64KB and then is broken up into 8-KB pages. Almost all I/O is performed in these sizes. Pages that are used in the same object are generally linked to one another to facilitate scanning. Pages are the building blocks that are used to hold all of the data and indexes that we take for granted when we write `SELECT * FROM TABLE` and all that wonderful code that has been written to translate that simple statement into a set of results starts churning to produce results quickly and correctly. How the data is actually formatted on page is based on a lot of factors: compression, partitioning, and most of all, indexing.

Indexing, like the entire gamut of performance-tuning topics, is hard to cover with any specificity on a written page (particularly not as a chapter of a larger book on design), I've given you some information about the mechanics of tables and indexes, and a few best practices, but to be realistic, it's never going to be enough without you working with a realistic, active working test system.

Tuning with indexes requires a lot of basic knowledge about structures and utilization of those structures applied on a large scale. Joins decide whether or not to use an index based on many factors, and the indexes available to a query affect the join operators chosen. The best teacher for this is the school of "having to wait for a five-hour query to process." Most people, when starting out, begin on small systems; they code any way they want and slap indexes on everything, and it works great. SQL Server has a very advanced optimizer that covers a multitude of such sins, particularly on a low-concurrency, low-usage system. As your system grows and requires more and more resources, it becomes more and more difficult to do performance tuning haphazardly, and all too often, programmers start applying indexes without actually seeing what effect they will have on the overall performance.

Designing physical structures is an important step to building high-performance systems that must be done during multiple phases of the project, starting when you are still modeling, and only being completed with performance testing, and honestly, continuing into production operation.

In the next chapter, we will talk a bit more about hardware as we look at some of the concerns for building highly concurrent systems with many users executing simultaneously. Having moderate- to high-level server-class hardware with fast RAID arrays and multiple channels to the disk storage, you can spread the data out over multiple filegroups, as I discussed in the basic table structure section earlier.

www.it-ebooks.info

CHAPTER 11

Coding for Concurrency

"It has been my observation that most people get ahead during the time that others waste."

—Henry Ford

Concurrency is all about having the computer utilize all of its resources simultaneously, or basically having more than one thing done at the same time when serving multiple users (technically, in SQL Server, you open multiple requests, on one or more connections). Even if you haven't done much with multiple users, if you know anything about computing you probably are familiar with the term *multitasking*. The key here is that when multiple processes or users are accessing the same resources, each user expects to see a consistent view of the data and certainly expects that other users will not be stomping on his or her results.

The topics of this chapter will center on understanding why and how you should write your database code or design your objects to make them accessible concurrently by as many users as you have in your system. In this chapter, I'll discuss the following:

- *OS and hardware concerns:* I'll briefly discuss various issues that are out of the control of SQL code but can affect concurrency.
- *Transactions:* I'll give an overview of how transactions work and how to start and stop them in T-SQL code.
- *SQL Server concurrency controls:* In this section, I'll explain locks and isolation levels that allow you to customize how isolated processes are from one another
- *Coding for concurrency:* I'll discuss methods of coding data access to protect from users simultaneously making changes to data and placing data into less-than-adequate situations. You'll also learn how to deal with users stepping on one another and how to maximize concurrency.

The key goal of this chapter is to acquaint you with many of the kinds of things SQL Server does to make it fast and safe to have multiple users doing the same sorts of tasks with the same resources and how you can optimize your code to make it easier for this to happen.

RESOURCE GOVERNOR

SQL Server 2008 had a new feature that is concurrency related (especially as it relates to performance tuning), though it is more of a management tool than a design concern. The feature is called Resource

www.it-ebooks.info

Governor, and it allows you to partition the workload of the entire server by specifying maximum and minimum resource allocations (memory, CPU, concurrent requests, etc.) to users or groups of users. You can classify users into groups using a simple user-defined function that, in turn, takes advantage of the basic server-level functions you have for identifying users and applications (`IS_SRVROLEMEMBER`, `APP_NAME`, `SYSTEM_USER`, etc.). Like many of the high-end features of SQL Server, Resource Governor is only available with the Enterprise Edition and higher editions.

Using Resource Governor, you can group together and limit the users of a reporting application, of Management Studio, or of any other application to a specific percentage of the CPU, a certain percentage and number of processors, and limited requests at one time. In SQL Server 2012, I/O limits are still not available in Resource Governor.

One nice thing about Resource Governor is that some settings can only apply when the server is under a load. So if the reporting user is the only active process, that user might get the entire server's power. But if the server is being heavily used, users would be limited to the configured amounts. I won't talk about Resource Governor anymore in this chapter, but it is definitely a feature that you might want to consider if you are dealing with different types of users in your applications.

What Is Concurrency?

The concept of concurrency can be boiled down to the following statement:

> *Maximize the amount of work that can be done by all users at the same time, and most importantly, make all users feel like they're important.*

Because of the need to balance the amount of work with the user's perception of the amount of work being done, there are going to be the following tradeoffs:

- *Number of concurrent users:* How many users can (or need to) be served at the same time?
- *Overhead:* How complex are the algorithms to maintain concurrency?
- *Accuracy:* How correct must the results be?
- *Performance:* How fast does each process finish?
- *Cost:* How much are you willing to spend on hardware and programming time?

As you can probably guess, if all the users of a database system never needed to run queries at the same time, life in database-system–design land would be far simpler. You would have no need to be concerned with what other users might want to do. The only real performance goal would be to run one process really fast and move to the next process. If no one ever shared resources, multitasking server operating systems would be unnecessary. All files could be placed on a user's local computer, and that would be enough. And if we could single-thread all activities on a server, more work might be done, but just like the old days, people would sit around waiting for their turns (yes, with mainframes, people actually did that sort of thing). Internally, the situation is still technically the same in a way, as a computer cannot process more individual instructions than it has cores in its CPUs, but it can run and swap around fast enough to make hundreds or thousands of people feel like they are the only users. This is especially true if your system engineer builds computers that are good as SQL Server machines (and not just file servers) and the architects/programmers build systems that meet the requirements for a relational database (and not just what seems expedient at the time).

www.it-ebooks.info

A common scenario for a multiuser database involves a sales and shipping application. You might have 50 salespeople in a call center trying to sell the last 25 closeout items that are in stock. It isn't desirable to promise the last physical item accidentally to multiple customers, since two users might happen to read that it was available at the same time and both be allowed to place an order for it. In this case, stopping the first order wouldn't be necessary, but you would want to disallow or otherwise prevent the second (or subsequent) orders from being placed, since they cannot be fulfilled immediately.

Most programmers instinctively write code to check for this condition and to try to make sure that this sort of thing doesn't happen. Code is generally written that does something along these lines:

- Check to make sure that there's adequate stock.
- Create a shipping row to allocate the product to the customer.

That's simple enough, but what if one person checks to see if the product is available at the same time as another, and more orders are placed than you have adequate stock for? This is a far more common possibility than you might imagine. Is this acceptable? If you've ever ordered a product that you were promised in two days and then found out your items are on backorder for a month, you know the answer to this question: "No! It is very unacceptable." When this happens, you try another retailer next time, right?

I should also note that the problems presented by concurrency aren't quite the same as those for *parallelism*, which is having one task split up and done by multiple resources at the same time. Parallelism involves a whole different set of problems and luckily is more or less not your problem. In writing SQL Server code, parallelism is done automatically, as tasks can be split among resources (sometimes, you will need to adjust just how many parallel operations can take place, but in practice, SQL Server does *most* of that work for you). When I refer to concurrency, I generally mean having multiple *different* operations happening at the same time by different connections to SQL Server. Here are just a few of the questions you have to ask yourself:

- What effect will there be if a query modifies rows that have already been used by a query in a different batch?
- What if the other query creates new rows that would have been important to the other batch's query? What if the other query deletes others?
- Most importantly, can one query corrupt another's results?

You must consider a few more questions as well. Just how important is concurrency to you, and how much are you willing to pay in performance? The whole topic of concurrency is basically a set of tradeoffs between performance, consistency, and the number of simultaneous users.

■ **Tip** Starting with SQL Server 2005, a new way to execute multiple batches of SQL code from the same connection simultaneously was added; it is known as Multiple Active Result Sets (MARS). It allows interleaved execution of several statements, such as `SELECT`, `FETCH`, `RECEIVE READTEXT`, or `BULK INSERT`. As the product continues to mature, you will start to see the term "request" being used in the place where we commonly thought of connection in SQL Server 2000 and earlier. Admittedly, this is still a hard change that has not yet become embedded in people's thought processes, but in some places (like in the Dynamic Management Views), you need to understand the difference.

MARS is principally a client technology and must be enabled by a connection, but it can change some of the ways that SQL Server handles concurrency. I'll note places where MARS affects the fundamentals of concurrency.

---

www.it-ebooks.info

OS and Hardware Concerns

SQL Server is designed to run on a variety of hardware types. Essentially the same basic code runs on a low-end netbook and on a clustered array of servers that rivals many supercomputers. Every machine running a version of SQL Server, from Express to Enterprise Edition, can have a vastly different concurrency profile. Each edition will also be able to support different amounts of hardware: Express supports 1GB of RAM and one processor socket (with up to 4 cores, which is still more than our first SQL Server that had 16MB of RAM), and at the other end of the spectrum, the Enterprise Edition can handle as much hardware as a manufacturer can stuff into one box. Additionally, a specialized version called the Parallel Data Warehouse edition is built specifically for data warehousing loads. The fact is that, in every version, many of the very same concerns exist concerning how SQL Server handles multiple users using the same resources seemingly simultaneously. Fast-forward to the future (i.e., now), and the Azure platform allows you to access your data in the cloud on massive computer systems from anywhere. In this section, I'll briefly touch on some of the issues governing concurrency that our T-SQL code needn't be concerned with, because concurrency is part of the environment we work in.

SQL Server and the OS balance all the different requests and needs for multiple users. It's beyond the scope of this book to delve too deeply into the details, but it's important to mention that concurrency is heavily tied to hardware architecture. For example, consider the following subsystems:

- *Processor:* The heart of the system is the CPU. It controls the other subsystems, as well as doing any calculations needed. If you have too few processors, excessive time can be wasted switching between requests.
- *Disk subsystem:* Disk is always the slowest part of the system (even with solid state drives becoming more and more prevalent). A slow disk subsystem is the downfall of many systems, particularly because of the expense involved. Each drive can read only one piece of information at a time, so to access disks concurrently, it's necessary to have multiple disk drives, and even multiple controllers or channels to disk drive arrays. Especially important is the choice between RAID systems, which take multiple disks and configure them for performance and redundancy:
    - *0:* Striping across all disks with no redundancy, performance only.
    - *1:* Mirroring between two disks, redundancy only.
        - *5:* Striping with distributed parity; excellent for reading, but can be slow for writing. Not typically suggested for most SQL Server OLTP usage, though it isn't horrible for lighter loads.
        - *0+1:* Mirrored stripes. Two RAID 1 arrays, mirrored. Great for performance, but not tremendously redundant.
        - *1+0 (also known as 10):* Striped mirrors. Some number of RAID 0 mirrored arrays, then striped across the mirrors. Usually, the best mix of performance and redundancy for an OLTP SQL Server installation.
- *Network interface:* Bandwidth to the users is critical but is usually less of a problem than disk access. However, it's important to attempt to limit the number of round trips between the server and the client. This is highly dependent on whether the client is connecting over a dialup connection or a gigabit Ethernet (or even multiple network interface cards). Turning on `SET NOCOUNT` in all connections and coded objects, such as stored procedures and triggers, is a good first step, because otherwise, a message is sent to the client for each query executed, requiring bandwidth (and processing) to deal with them.

www.it-ebooks.info

- *Memory:* One of the cheapest commodities that you can improve substantially on a computer is memory. SQL Server 2012 can use a tremendous amount of memory within the limits of the edition you used (and the amount of RAM will not affect your licensing costs like processor cores either.)

Each of these subsystems needs to be in balance to work properly. You could theoretically have 100 CPUs and 128GB of RAM, and your system could still be slow. In this case, a slow disk subsystem could be causing your issues. The goal is to maximize utilization of *all* subsystems—the faster the better—but it's useless to have super-fast CPUs with a super-slow disk subsystem. Ideally, as your load increases, disk, CPU, and memory usage would increase proportionally, though this is a heck of a hard thing to do. The bottom line is that the number of CPUs, disk channels, disk drives, and network cards and the amount of RAM you have all affect concurrency.

Monitoring hardware and OS performance issues is a job primarily for `perfmon` and/or the data collector. Watching counters for CPU, memory, SQL Server, and so on lets you see the balance among all the different subsystems. In the end, poor hardware configuration can kill you just as quickly as poor SQL Server implementation.

For the rest of this chapter, I'm going to ignore these types of issues and leave them to others with a deeper hardware focus, such as the MSDN web site (`http://msdn.microsoft.com`) or great blogs like Glenn Berry's (`http://sqlserverperformance.wordpress.com`). I'll be focusing on design- and coding-related issues pertaining to how to code better SQL to manage concurrency between SQL Server processes.

Transactions

No discussion of concurrency can really have much meaning without an understanding of the transaction. *Transactions* are a mechanism that allows one or more statements to be guaranteed either to be fully completed or to fail totally. It is an internal SQL Server mechanism that is used to keep the data that's written to and read from tables consistent throughout a batch, as required by the user.

Whenever data is modified in the database, the changes are not written to the physical data files, but first to a page in RAM and then a log of every change is written to the transaction log immediately before the change is registered as complete. (Any log files need to be on a very fast disk drive subsystem for this reason). Later, the physical table structure is written to when the system is able to do the write (during what is called a *checkpoint*). Understanding the process of how modifications to data are made is essential, because while tuning your overall system, you have to be cognizant that every modification operation is logged when considering how large to make your transaction log.

The purpose of transactions is to provide a mechanism to allow multiple processes access to the same data simultaneously, while ensuring that logical operations are either carried out entirely or not at all. To explain the concurrency issues that transactions help with, there's a common acronym: ACID. It stands for the following:

- *Atomicity:* Every operation within a transaction is treated as a singular operation; either all its data modifications are performed, or none of them is performed.
- *Consistency:* Once a transaction is completed, the system must be left in a consistent state. This means that all the constraints on the data that are part of the RDBMS definition must be honored.
- *Isolation:* This means that the operations within a transaction must be suitably isolated from other transactions. In other words, no other transactions ought to see data in an intermediate state, within the transaction, until it's finalized. This is typically done by using locks (for details on locks, refer to the section "SQL Server Concurrency Controls" later in this chapter).
- *Durability:* Once a transaction is completed (committed), all changes must be persisted as requested. The modifications should persist even in the event of a system failure.

www.it-ebooks.info

Transactions are used in two different ways. The first way is to provide for isolation between processes. Every DML and DDL statement, including `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, `ALTER TABLE`, `CREATE INDEX`, and even `SELECT` statements, that is executed in SQL Server is run within a transaction. If you are in the middle of adding a column to the table, you don't want another user to try to modify data in the table at the same time. For DDL and modification statements, such as `INSERT`, `UPDATE`, and `DELETE`, locks are placed, and all system changes are recorded in the transaction log. If any operation fails, or if the user asks for an operation to be undone, SQL Server uses the transaction log to undo the operations already performed. For `SELECT` operations (and during the selection of rows to modify/remove from a table), locks will also be used to ensure that data isn't changed as it is being read.

Second, the programmer can use transaction commands to batch together multiple commands into one logical unit of work. For example, if you write data to one table successfully, and then try unsuccessfully to write to another table, the initial writes can be undone. This section will mostly be about defining and demonstrating this syntax.

The key to using transactions is that, when writing statements to modify data using one or more SQL statements, you need to make use of transactions to ensure that data is written safely and securely. A typical problem with procedures and operations in T-SQL code is underusing transactions, so that when unexpected errors (such as security problems, constraint failures, and hardware glitches) occur, orphaned or inconsistent data is the result. And when, a few weeks later, users are complaining about inconsistent results, you have to track down the issues; you lose some sleep; and your client loses confidence in the system you have created and more importantly loses confidence in you.

How long the log is stored is based on the recovery model under which your database is operating. There are three models:

- *Simple:* The log is maintained only until the operation is executed and a checkpoint is executed (by SQL Server automatically or manually). A checkpoint operation makes certain that the data has been written to the data files, so it is permanently stored.
- *Full:* The log is maintained until you explicitly clear it out.
- *Bulk logged:* This keeps a log much like the full recovery model but doesn't fully log some operations, such as `SELECT INTO`, bulk loads, index creations, or text operations. It just logs that the operation has taken place. When you back up the log, it will back up extents that were added during `BULK` operations, so you get full protection with quicker bulk operations.

Even in the simple model, you must be careful about log space, because if large numbers of changes are made in a single transaction or very rapidly, the log rows must be stored at least until all transactions are committed and a checkpoint takes place. This is clearly just a taste of transaction log management; for a more complete explanation, please see SQL Server 2012 Books Online.

Transaction Syntax

The syntax to start and stop transactions is pretty simple. I'll cover four variants of the transaction syntax in this section:

- *Basic transactions:* The syntax of how to start and complete a transaction
- *Nested transactions:* How transactions are affected when one is started when another is already executing
- *Savepoints:* Used to selectively cancel part of a transaction
- *Distributed transactions:* Using transactions to control saving data on multiple SQL Servers

In the final part of this section, I'll also cover explicit versus implicit transactions. These sections will give you the foundation needed to move ahead and start building proper code, ensuring that each modification is done properly, even when multiple SQL statements are necessary to form a single-user operation.

www.it-ebooks.info

Basic Transactions

In transactions' basic form, three commands are required: `BEGIN TRANSACTION` (to start the transaction), `-COMMIT TRANSACTION` (to save the data), and `ROLLBACK TRANSACTION` (to undo the changes that were made). It's as simple as that.

For example, consider the case of building a stored procedure to modify two tables. Call these tables `table1` and `table2`. You'll modify `table1`, check the error status, and then modify `table2` (these aren't real tables, just syntax examples):

```
BEGIN TRY
  BEGIN TRANSACTION;
      UPDATE table1
      SET value = 'value';

      UPDATE table2
      SET value = 'value';
  COMMIT TRANSACTION;
END TRY
BEGIN CATCH
      ROLLBACK TRANSACTION;
      THROW 50000,'An error occurred',16
END CATCH
```

Now, if some unforeseen error occurs while updating either `table1` or `table2`, you won't get into the case where `table1` is updated and `table2` is not. It's also imperative not to forget to close the transaction (either save the changes with `COMMIT TRANSACTION`, or undo the changes with `ROLLBACK TRANSACTION`), because the open transaction that contains your work is in a state of limbo, and if you don't either complete it or roll it back, it can cause a lot of issues just hanging around in an open state. For example, if the transaction stays open and other operations are executed within that transaction, you might end up losing all work done on that connection. You may also prevent other connections from getting their work done, because each connection is isolated from one another messing up or looking at their unfinished work. Another user who needed the affected rows in `table1` or `table2` would have to wait (more on why this is throughout this chapter). The worst case of this I saw a number of years back was a single connection that was open all day with a transaction open after a failure because there was no error handling on the transaction. We lost a day's work because we finally had to roll back the transactions when we killed the process.

There's an additional setting for simple transactions known as *named transactions*, which I'll introduce for completeness. (Ironically, this explanation will take more ink than introducing the more useful transaction syntax, but it is something good to know and can be useful in rare circumstances!) You can extend the functionality of transactions by adding a transaction name, as shown:

```
BEGIN TRANSACTION <tranName> or <@tranvariable>;
```

This can be a confusing extension to the `BEGIN TRANSACTION` statement. It names the transaction to make sure you roll back to it, for example:

```
BEGIN TRANSACTION one;
ROLLBACK TRANSACTION one;
```

Only the first transaction mark is registered in the log, so the following code returns an error:

```
BEGIN TRANSACTION one;
BEGIN TRANSACTION two;
ROLLBACK TRANSACTION two;
```


www.it-ebooks.info

The error message is as follows:

```
Msg 6401, Level 16, State 1, Line 3
Cannot roll back two. No transaction or savepoint of that name was found.
```

Unfortunately, after this error has occurred, the transaction is still left open. For this reason, it's seldom a good practice to use named transactions in your code unless you have a very specific purpose. The specific use that makes named transactions interesting is when named transactions use the `WITH MARK` setting. This allows marking the transaction log, which can be used when restoring a transaction log instead of a date and time. A common use of the marked transaction is to restore several databases back to the same condition and then restore all of the databases to a -common mark.

■ **Note** A very good practice in testing your code that deals with transactions is to make sure that there are no transactions by executing `ROLLBACK TRANSACTION` until the message "`The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION`" is returned. At that point, you can feel safe that you are outside a transaction. In code, you should use `@@trancount` to check, which I will demonstrate later in this chapter.

The mark is only registered if data is modified within the transaction. A good example of its use might be to build a process that marks the transaction log every day before some daily batch process, especially one where the database is in single-user mode. The log is marked, and you run the process, and if there are any troubles, the database log can be restored to just before the mark in the log, no matter when the process was executed. Using the `AdventureWorks2012` database, I'll demonstrate this capability. You can do the same, but be careful to do this somewhere where you know you have a proper backup (just in case something goes wrong).

We first set up the scenario by putting the `AdventureWorks2012` database in full recovery model.

```
USE Master;
GO

ALTER DATABASE AdventureWorks2012
      SET RECOVERY FULL;
```

Next, we create a couple of backup devices to hold the backups we're going to do:

```
EXEC sp_addumpdevice 'disk', 'TestAdventureWorks2012',
                             'C:\SQL\Backup\AdventureWorks2012.bak';
EXEC sp_addumpdevice 'disk', 'TestAdventureWorks2012Log',
                             'C:\SQL\Backup\AdventureWorks2012Log.bak';
```

■ **Tip** You can see the current setting using the following code:

```
SELECT recovery_model_desc
FROM   sys.databases
WHERE  name = 'AdventureWorks2012';
```


www.it-ebooks.info

Next, we back up the database to the dump device we created:

```
BACKUP DATABASE AdventureWorks2012 TO TestAdventureWorks2012;
```

Now, we change to the AdventureWorks2012 database and delete some data from a table:

```
USE AdventureWorks2012;
GO
SELECT COUNT(*)
FROM   Sales.SalesTaxRate;

BEGIN TRANSACTION Test WITH MARK 'Test';
DELETE Sales.SalesTaxRate;
COMMIT TRANSACTION;
```

This returns 29. Run the SELECT statement again, and it will return 0. Next, back up the transaction log to the other backup device:

```
BACKUP LOG AdventureWorks2012 to TestAdventureWorks2012Log;
```

Now, we can restore the database using the RESTORE DATABASE command (the NORECOVERY setting keeps the database in a state ready to add transaction logs). We apply the log with RESTORE LOG. For the example, we'll only restore up to before the mark that was placed, not the entire log:

```
USE Master
GO
RESTORE DATABASE AdventureWorks2012 FROM TestAdventureWorks2012
                                              WITH REPLACE, NORECOVERY;

RESTORE LOG AdventureWorks2012 FROM TestAdventureWorks2012Log
                                              WITH STOPBEFOREMARK = 'Test', RECOVERY;
```

Now, execute the counting query again, and you can see that the 29 rows are in there.

```
USE AdventureWorks2012;
GO
SELECT COUNT(*)
FROM   Sales.SalesTaxRate;
```

If you wanted to include the actions within the mark, you could use STOPATMARK instead of STOPBEFOREMARK. You can find the log marks that have been made in the MSDB database in the -logmarkhistory table.

Nesting Transactions

Yes, I am aware that the title of this section probably sounds a bit like Marlin Perkins is going to take over and start telling of the mating habits of transactions, but I am referring to starting a transaction after another transaction has already been started. The fact is that you can nest the starting of transactions like this:

```
BEGIN TRANSACTION;
    BEGIN TRANSACTION;
       BEGIN TRANSACTION;
```

Technically speaking, there is really only one transaction being started, but an internal counter is keeping up with how many logical transactions have been started. To commit the transactions, you have to execute the same number of COMMIT TRANSACTION commands as the number of BEGIN TRANSACTION commands that have been executed. To tell how many BEGIN TRANSACTION commands have been executed without being committed,

www.it-ebooks.info

you can use the @@TRANCOUNT global variable. When it's equal to one, then one BEGIN TRANSACTION has been executed. If it's equal to two, then two have, and so on. When @@TRANCOUNT equals zero, you are no longer within a transaction context.

The limit to the number of transactions that can be nested is extremely large (the limit is 2,147,483,647, which took about 1.75 hours to reach in a tight loop on my old 2.27-GHz laptop with 2GB of RAM—clearly far, far more than *any* process should ever need).

As an example, execute the following:

```
SELECT @@TRANCOUNT AS zeroDeep;
BEGIN TRANSACTION;
SELECT @@TRANCOUNT AS oneDeep;
```

It returns the following results:

```
zeroDeep
--------
0

oneDeep
-------
1
```

Then, nest another transaction, and check @@TRANCOUNT to see whether it has incremented. Afterward, commit that transaction, and check @@TRANCOUNT again:

```
BEGIN TRANSACTION;
SELECT @@TRANCOUNT AS twoDeep;
COMMIT TRANSACTION; --commits very last transaction started with BEGIN TRANSACTION
SELECT @@TRANCOUNT AS oneDeep;
```

This returns the following results:

```
twoDeep
-------
2

oneDeep
-------
1
```

Finally, close the final transaction:

```
COMMIT TRANSACTION;
SELECT @@TRANCOUNT AS zeroDeep;
```

This returns the following result:

```
zeroDeep
--------
0
```


www.it-ebooks.info

As I mentioned earlier in this section, technically only one transaction is being started. Hence, it only takes one ROLLBACK TRANSACTION command to roll back as many transactions as you have nested. So, if you've coded up a set of statements that end up nesting 100 transactions and you issue one rollback transaction, all transactions are rolled back—for example:

```
BEGIN TRANSACTION;
BEGIN TRANSACTION;
BEGIN TRANSACTION;
BEGIN TRANSACTION;
BEGIN TRANSACTION;
BEGIN TRANSACTION;
BEGIN TRANSACTION;
SELECT @@trancount AS InTran;

ROLLBACK TRANSACTION;
SELECT @@trancount AS OutTran;
```

This returns the following results:

```
InTran
-------
7

OutTran
-------
0
```

This is, by far, the trickiest part of using transactions in your code leading to some messy error handling and code management. It's a bad idea to just issue a ROLLBACK TRANSACTION command without being cognizant of what will occur once you do—especially the command's influence on the following code. If code is written expecting to be within a transaction and it isn't, your data can get corrupted.

In the preceding example, if an UPDATE statement had been executed immediately after the ROLLBACK command, it wouldn't be executed within an explicit transaction. Also, if COMMIT TRANSACTION is executed immediately after the ROLLBACK command, an error will occur:

```
SELECT @@trancount
COMMIT TRANSACTION
```

This will return

```
-----------
0

Msg 3902, Level 16, State 1, Line 2
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

Savepoints

In the previous section, I explained that all open transactions are rolled back using a ROLLBACK TRANSACTION call. This isn't always desirable, so a tool is available to roll back only certain parts of a transaction. Unfortunately, it requires forethought and a special syntax. *Savepoints* are used to provide "selective" rollback.

www.it-ebooks.info

For this, from within a transaction, issue the following statement:

```
SAVE TRANSACTION <savePointName>; --savepoint names must follow the same rules for
                                  --identifiers as other objects
```

For example, use the following code in whatever database you desire. In the source code, I'll continue to place it in the tempdb, because the examples are self-contained.

```
CREATE SCHEMA arts;
GO
CREATE TABLE arts.performer
(
    performerId int NOT NULL IDENTITY,
    name varchar(100) NOT NULL
 );
GO
BEGIN TRANSACTION;
INSERT INTO arts.performer(name) VALUES ('Elvis Costello');

SAVE TRANSACTION savePoint;

INSERT INTO arts.performer(name) VALUES ('Air Supply');

--don't insert Air Supply, yuck! …
ROLLBACK TRANSACTION savePoint;

COMMIT TRANSACTION;

SELECT *
FROM arts.performer;
```

The output of this listing is as follows:

```
performerId     name
-----------     --------------
1               Elvis Costello
```

In the code, there were two INSERT statements within the transaction boundaries, but in the output, there's only one row. Obviously, the row that was rolled back to the savepoint wasn't persisted.

Note that you don't commit a savepoint; SQL Server simply places a mark in the transaction log to tell itself where to roll back to if the user asks for a rollback to the savepoint. The rest of the operations in the overall transaction aren't affected. Savepoints don't affect the value of @@trancount, nor do they release any locks that might have been held by the operations that are rolled back, until all nested transactions have been committed or rolled back.

Savepoints give the power to effect changes on only part of the operations transaction, giving you more control over what to do if you're deep in a large number of operations.

I'll mention savepoints later in this chapter when writing stored procedures, as they allow the rolling back of all the actions of a single stored procedure without affecting the transaction state of the stored procedure caller, though in most cases, it is usually just easier to roll back the entire transaction. Savepoints do, however, allow you to perform some operation, check to see if it is to your liking, and if it's not, roll it back.

You can't use savepoints in a couple situations:

- When you're using MARS and executing more than one batch at a time
- When the transaction is enlisted into a distributed transaction (the next section discusses this)

www.it-ebooks.info

HOW MARS AFFECTS TRANSACTIONS

There's a slight wrinkle in how multiple statements can behave when using OLE DB or ODBC native client drivers to retrieve rows in SQL Server 2005 or later. When executing batches under MARS, there can be a couple scenarios:

- *Connections set to automatically commit*: Each executed batch is within its own transaction, so there are multiple transaction contexts on a single connection.
- *Connections set to be manually committed*: All executed batches are part of one transaction.

When MARS is enabled for a connection, any batch or stored procedure that starts a transaction (either implicitly in any statement or by executing `BEGIN TRANSACTION`) must commit the transaction; if not, the transaction will be rolled back. These transactions were new to SQL Server 2005 and are referred to as batch-scoped transactions.

Distributed Transactions

It would be wrong not to at least bring up the subject of distributed transactions. Occasionally, you might need to update data on a server that's different from the one on which your code resides. The Microsoft Distributed Transaction Coordinator service (MS DTC) gives us this ability.

If your servers are running the MS DTC service, you can use the `BEGIN DISTRIBUTED TRANSACTION` command to start a transaction that covers the data residing on your server, as well as the remote server. If the server configuration `'remote proc trans'` is set to `1`, any transaction that touches a linked server will start a distributed transaction without actually calling the `BEGIN DISTRIBUTED TRANSACTION` command. However, I would strongly suggest you know if you will be using another server in a transaction (check `sys.configurations` or `sp_configure` for the current setting, and set the value using `sp_configure`). Note also that savepoints aren't supported for distributed transactions.

The following code is just pseudocode and won't run as is, but this is representative of the code needed to do a distributed transaction:

```
BEGIN TRY
    BEGIN DISTRIBUTED TRANSACTION;

    --remote server is a server set up as a linked server

    UPDATE remoteServer.dbName.schemaName.tableName
    SET value = 'new value'
    WHERE keyColumn = 'value';

    --local server
    UPDATE dbName.schemaName.tableName
    SET value = 'new value'
    WHERE keyColumn = 'value';

    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    DECLARE @ERRORMessage varchar(2000);
    SET @ERRORMessage = ERROR_MESSAGE();
    THROW 50000, @ERRORMessage,16;
END CATCH
```


www.it-ebooks.info

The distributed transaction syntax also covers the local transaction. As mentioned, setting the configuration option `'remote proc trans'` automatically upgrades a `BEGIN TRANSACTION` command to a `BEGIN DISTRIBUTED TRANSACTION` command. This is useful if you frequently use distributed transactions. Without this setting, the remote command is executed, but it won't be a part of the current -transaction.

Explicit vs. Implicit Transactions

Before finishing the discussion of transaction syntax, there's one last thing that needs to be covered for the sake of completeness. I've alluded to the fact that every statement is executed in a transaction (again, this includes even `SELECT` statements). This is an important point that must be understood when writing code. Internally, SQL Server starts a transaction every time a SQL statement is started. Even if a transaction isn't started explicitly with a `BEGIN TRANSACTION` statement, SQL Server automatically starts a new transaction whenever a statement starts and commits or rolls it back depending on whether or not any errors occur. This is known as an *autocommit* transaction. When the SQL Server engine commits the transaction, it starts for each statement-level transaction.

SQL Server gives us a setting to change this behavior of automatically committing the transaction: `SET IMPLICIT_TRANSACTIONS`. When this setting is turned on and the execution context isn't already within a transaction, such as one explicitly declared using `BEGIN TRANSACTION`, `BEGIN TRANSACTION` is automatically (logically) executed when any of the following statements are executed: `INSERT`, `UPDATE`, `DELETE`, `SELECT`, `TRUNCATE TABLE`, `DROP`, `ALTER TABLE`, `REVOKE`, `CREATE`, `GRANT`, `FETCH`, or `OPEN`. This will mean that a `COMMIT TRANSACTION` or `ROLLBACK TRANSACTION` command has to be executed to end the transaction. Otherwise, once the connection terminates, all data is lost (and until the transaction terminates, locks that have been accumulated are held, other users are blocked, and pandemonium might occur).

`SET IMPLICIT_TRANSACTIONS` isn't a typical setting used by SQL Server programmers or administrators but is worth mentioning because if you change the setting of `ANSI_DEFAULTS` to `ON`, `IMPLICIT_TRANSACTIONS` will be enabled!

I've mentioned that every `SELECT` statement is executed within a transaction, but this deserves a bit more explanation. The entire process of rows being considered for output, then transporting them from the server to the client is contained inside a transaction. The `SELECT` statement isn't finished until the entire result set is exhausted (or the client cancels the fetching of rows), so the transaction doesn't end either. This is an important point that will come back up in the "Isolation Levels" section, as I discuss how this transaction can seriously affect concurrency based on how isolated you need your queries to be.

Compiled SQL Server Code

Now that I've discussed the basics of transactions, it's important to understand some of the slight differences involved in using them within compiled code versus the way we have used transactions so far in batches. You can't use transactions in user-defined functions (you can't change system state in a function, so they aren't necessary anyhow), but it is important to understand the caveats when you use them in

- Stored procedures
- Triggers

Stored Procedures

Stored procedures, simply being compiled batches of code, use transactions as previously discussed, with one caveat. The transaction nesting level cannot be affected during the execution of a procedure. In other words, you must commit at least as many transactions as you begin in a stored procedure if you want everything to behave smoothly.

www.it-ebooks.info

Although you can roll back any transaction, you shouldn't roll it back unless the @@TRANCOUNT was zero when the procedure started. However, it's better not to execute a ROLLBACK TRANSACTION statement at all in a stored procedure, so there's no chance of rolling back to a transaction count that's different from when the procedure started. This protects you from the situation where the procedure is executed in another transaction. Rather, it's generally best to start a transaction and then follow it with a savepoint. Later, if the changes made in the procedure need to be backed out, simply roll back to the savepoint, and commit the transaction. It's then up to the stored procedure to signal to any caller that it has failed and to do whatever it wants with the transaction.

As an example, let's build the following simple procedure that does nothing but execute a BEGIN TRANSACTION and a ROLLBACK TRANSACTION:

```
CREATE PROCEDURE tranTest
AS
BEGIN
  SELECT @@TRANCOUNT AS trancount;

  BEGIN TRANSACTION;
  ROLLBACK TRANSACTION;
END;
```

Execute this procedure outside a transaction, and you will see it behaves like you would expect:

```
EXECUTE tranTest;
```

It returns

```
Trancount
---------
0
```

However, say you execute it within an existing transaction:

```
BEGIN TRANSACTION;

EXECUTE tranTest;
COMMIT TRANSACTION;
```

The procedure returns the following results:

```
Trancount
---------
1

Msg 266, Level 16, State 2, Procedure tranTest, Line 0
Transaction count after EXECUTE indicates a mismatching number of BEGIN and COMMIT statements.
Previous count = 1, current count = 0.

Msg 3902, Level 16, State 1, Line 3
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

The errors occur because the transaction depth has changed while rolling back the transaction inside the procedure. This error is one of the most frightening errors out there, because it usually says that you probably have been doing work that you expected to be *in* a transaction outside of a transaction, thus you will probably end up with out-of-sync data that needs to be repaired.

www.it-ebooks.info

Finally, let's recode the procedure as follows, putting a savepoint name on the transaction so we only roll back the code in the procedure:

```
ALTER PROCEDURE tranTest
AS
BEGIN
  --gives us a unique savepoint name, trim it to 125 characters if the
  --user named the procedure really really large, to allow for nestlevel
  DECLARE @savepoint nvarchar(128) =
      CAST(OBJECT_NAME(@@procid) AS nvarchar(125)) +
                        CAST(@@nestlevel AS nvarchar(3));

  SELECT @@TRANCOUNT AS trancount;
  BEGIN TRANSACTION;
  SAVE TRANSACTION @savepoint;
    --do something here
  ROLLBACK TRANSACTION @savepoint;
  COMMIT TRANSACTION;
END;
```

Now, you can execute it from within any number of transactions, and it will never fail, but it will never actually do anything either:

```
BEGIN TRANSACTION;
EXECUTE tranTest;
COMMIT TRANSACTION;
```

Now, it returns

```
Trancount
---------
1
```

You can call procedures from other procedures (even recursively from the same procedure) or external programs. It's important to take these precautions to make sure that the code is safe under any calling circumstances.

■ **Caution** As mentioned in the "Savepoints" section, you can't use savepoints with distributed transactions or when sending multiple batches over a MARS–enabled connection. To make the most out of MARS, you might not be able to use this strategy. Frankly speaking, it might simply be prudent to execute modification procedures one at a time anyhow.

Naming savepoints is important. Because savepoints aren't scoped to a procedure, you must ensure that they're always unique. I tend to use the procedure name (retrieved here by using the `object_name` function called for the `@@procId`, but you could just enter it textually) and the current transaction nesting level. This guarantees that I can never have the same savepoint active, even if calling the same procedure recursively.

Let's look briefly at how to code this into procedures using proper error handling:

www.it-ebooks.info

```
ALTER PROCEDURE tranTest
AS
BEGIN
  --gives us a unique savepoint name, trim it to 125
  --characters if the user named it really large
  DECLARE @savepoint nvarchar(128) =
              CAST(OBJECT_NAME(@@procid) AS nvarchar(125)) +
                                      CAST(@@nestlevel AS nvarchar(3));
  --get initial entry level, so we can do a rollback on a doomed transaction
  DECLARE @entryTrancount int = @@trancount;

  BEGIN TRY
    BEGIN TRANSACTION;
    SAVE TRANSACTION @savepoint;

    --do something here
    THROW 50000, 'Invalid Operation',16;

    COMMIT TRANSACTION;
  END TRY
  BEGIN CATCH

    --if the tran is doomed, and the entryTrancount was 0,
    --we have to roll back
      IF XACT_STATE()= -1 and @entryTrancount = 0
          ROLLBACK TRANSACTION;
      --otherwise, we can still save the other activities in the
      --transaction.
      ELSE IF XACT_STATE() = 1 --transaction not doomed, but open
         BEGIN
           ROLLBACK TRANSACTION @savepoint;
           COMMIT TRANSACTION;
         END

      DECLARE @ERRORmessage nvarchar(4000);
      SET @ERRORmessage = 'Error occurred in procedure ''' + OBJECT_NAME(@@procid)
                          + ''', Original Message: ''' + ERROR_MESSAGE() + '''';
      THROW 50000, @ERRORmessage,16;
      RETURN -100
  END CATCH
END
```

In the CATCH block, instead of rolling back the transaction, I checked for a doomed transaction, and if we were not in a transaction at the start of the procedure, I rolled back. A *doomed transaction* is one in which some operation has made it impossible to do anything other than roll back the transaction. A common cause is a trigger-based error message. It is still technically an active transaction, giving you the chance to roll it back so operations do occur outside of the expected transaction space.

If the transaction was not doomed, I simply rolled back the savepoint. An error is returned for the caller to deal with. You could also eliminate RAISERROR altogether if the error wasn't critical and the caller needn't ever know of the rollback. You can place any form of error handling in the CATCH block, and as long as you don't roll back the entire transaction and the transaction does not become doomed, you can keep going and later commit the transaction.

If this procedure called another procedure that used the same error handling, it would roll back its part of the transaction. It would then raise an error, which, in turn, would cause the CATCH block to be called, roll back

www.it-ebooks.info

the savepoint, and commit the transaction (at that point, the transaction wouldn't contain any changes at all). If the transaction is doomed, when you get to the top level, it is rolled back. You might ask why you should go through this exercise if you're just going to roll back the transaction anyhow. The key is that each level of the calling structure can decide what to do with its part of the transaction. Plus, in the error handler we have created, we get the basic call stack for debugging purposes.

As an example of how this works, consider the following schema and table (create it in any database you desire, likely tempdb, as this sample is isolated to this section):

```
CREATE SCHEMA menu;
GO
CREATE TABLE menu.foodItem
(
    foodItemId int NOT NULL IDENTITY(1,1)
        CONSTRAINT PKmenu_foodItem PRIMARY KEY,
    name varchar(30) NOT NULL
        CONSTRAINT AKmenu_foodItem_name UNIQUE,
    description varchar(60) NOT NULL,
        CONSTRAINT CHKmenu_foodItem_name CHECK (name <> ''),
        CONSTRAINT CHKmenu_foodItem_description CHECK (description <> '')
);
```

Now, create a procedure to do the insert:

```
CREATE PROCEDURE menu.foodItem$insert
(
    @name varchar(30),
    @description varchar(60),
    @newFoodItemId int = NULL OUTPUT --we will send back the new id here
)
AS
BEGIN
  SET NOCOUNT ON;

  --gives us a unique savepoint name, trim it to 125
  --characters if the user named it really large
  DECLARE @savepoint nvarchar(128) =
              CAST(OBJECT_NAME(@@procid) AS nvarchar(125)) +
                                    CAST(@@nestlevel AS nvarchar(3));
  --get initial entry level, so we can do a rollback on a doomed transaction
  DECLARE @entryTrancount int = @@trancount;

  BEGIN TRY
    BEGIN TRANSACTION;
    SAVE TRANSACTION @savepoint;

    INSERT INTO menu.foodItem(name, description)
    VALUES (@name, @description);

    SET @newFoodItemId = SCOPE_IDENTITY(); --if you use an instead of trigger,
                                           --you will have to use name as a key
                                           --to do the identity "grab" in a SELECT
                                           --query
    COMMIT TRANSACTION;
  END TRY
```


www.it-ebooks.info

```
  BEGIN CATCH

  --if the tran is doomed, and the entryTrancount was 0,
  --we have to roll back
    IF XACT_STATE()= -1 and @entryTrancount = 0
       ROLLBACK TRANSACTION;
    --otherwise, we can still save the other activities in the
    --transaction.
    ELSE IF XACT_STATE() = 1 --transaction not doomed, but open
       BEGIN
         ROLLBACK TRANSACTION @savepoint;
         COMMIT TRANSACTION;
       END

    DECLARE @ERRORmessage nvarchar(4000);
    SET @ERRORmessage = 'Error occurred in procedure ''' + object_name(@@procid)
                        + ''', Original Message: ''' + ERROR_MESSAGE() + '''';
    --change to RAISERROR (50000, @ERRORmessage,16) if you want to continue processing
    THROW 50000,@ERRORmessage, 16;

    RETURN -100;
  END CATCH
END;
```

Next, try out the code:

```
DECLARE @foodItemId int, @retval int;
EXECUTE @retval = menu.foodItem$insert @name ='Burger',
                                       @description = 'Mmmm Burger',
                                       @newFoodItemId = @foodItemId output;
SELECT @retval AS returnValue;
IF @retval >= 0
    SELECT foodItemId, name, description
    FROM   menu.foodItem
    where  foodItemId = @foodItemId;
```

There's no error, so the row we created is returned:

```
returnValue
-----------
0

foodItemId      name    description
----------      ----    -----------
1 Burger        Mmmm    Burger
```

Now, try out the code with an error:

```
DECLARE @foodItemId int, @retval int;
EXECUTE @retval = menu.foodItem$insert @name ='Big Burger',
                                       @description = '',
                                       @newFoodItemId = @foodItemId output;
SELECT @retval AS returnValue;
```


www.it-ebooks.info

```
IF @retval >= 0
    SELECT foodItemId, name, description
    FROM menu.foodItem
    where foodItemId = @foodItemId;
```

Because the description is blank, an error is returned:

```
Msg 50000, Level 16, State 16, Procedure foodItem$insert, Line 50
Error occurred in procedure 'foodItem$insert', Original Message: 'The INSERT statement
conflicted with the CHECK constraint "CHKmenu_foodItem_description". The conflict occurred in
database "ContainedDatabase", table "menu.foodItem", column 'description'.'
```

Note that no code in the batch is executed after the THROW statement is executed. Using RAISERROR will allow the processing to continue if you so desire.

Triggers

Just as in stored procedures, you can start transactions, set savepoints, and roll back to a savepoint. However, if you execute a ROLLBACK TRANSACTION statement in a trigger, two things can occur:

- Outside a TRY-CATCH block, the entire batch of SQL statements is canceled.
- Inside a TRY-CATCH block, the batch isn't canceled, but the transaction count is back to zero.

Back in Chapters 6 and 7, we discussed and implemented triggers that consistently used rollbacks when any error occurred. If you're not using TRY-CATCH blocks, this approach is generally exactly what's desired, but when using TRY-CATCH blocks, it can make things more tricky. To handle this, in the CATCH block of stored procedures I've included this code:

```
--if the tran is doomed, and the entryTrancount was 0,
--we have to roll back
  IF XACT_STATE()= -1 and @entryTrancount = 0
      ROLLBACK TRANSACTION;
  --otherwise, we can still save the other activities in the
  --transaction.
  ELSE IF XACT_STATE() = 1 --transaction not doomed, but open
     BEGIN
       ROLLBACK TRANSACTION @savepoint;
       COMMIT TRANSACTION;
     END
```

This is an effective, if perhaps limited, method of working with errors from triggers that works in most any situation. Removing all ROLLBACK TRANSACTION commands but just raising an error from a trigger dooms the transaction, which is just as much trouble as the rollback. The key is to understand how this might affect the code that you're working with and to make sure that errors are handled in an understandable way. More than anything, test all types of errors in your system (trigger, constraint, and so on).

For an example, I will create a trigger based on the framework we used for triggers in Chapter 6 and 7, which is presented in more detail in Appendix B. Instead of any validations, I will just immediately cause an error with the statement THROW 50000,'FoodItem''s cannot be done that way',16. Note that my trigger template does do a rollback in the trigger, assuming that users of these triggers follow the error handing setup here, rather than just dooming the transaction. Dooming the transaction could be a safer way to go if you do not have full control over error handling.

www.it-ebooks.info

```
CREATE TRIGGER menu.foodItem$InsertTrigger
ON menu.foodItem
AFTER INSERT
AS
BEGIN
  SET NOCOUNT ON;
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000), --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
    @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  --@rowsAffected int = (SELECT COUNT(*) FROM deleted);
  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  BEGIN TRY
      --[validation blocks][validation section]
  THROW 50000, 'FoodItem''s cannot be done that way',16
      --[modification blocks][modification section]
  END TRY

  BEGIN CATCH
            IF @@TRANCOUNT > 0
                ROLLBACK TRANSACTION;

            THROW;
    END CATCH
END
```

In the downloadable code, I have modified the error handling in the stored procedure to put out markers, so you can see what branch of the code is being executed:

```
    SELECT 'In error handler'

    --if the tran is doomed, and the entryTrancount was 0,
    --we have to roll back
    IF XACT_STATE()= -1 and @entryTrancount = 0
      BEGIN

         SELECT 'Transaction Doomed'

         ROLLBACK TRANSACTION
       END
     --otherwise, we can still save the other activities in the
    --transaction.
    ELSE IF XACT_STATE() = 1 --transaction not doomed, but open
       BEGIN
         SELECT 'Savepoint Rollback'
```


www.it-ebooks.info

```
        ROLLBACK TRANSACTION @savepoint
        COMMIT TRANSACTION
    END
```

Executing the code that contains an error that the constraints catch:

```
DECLARE @foodItemId int, @retval int;
EXECUTE @retval = menu.foodItem$insert @name ='Big Burger',
                                       @description = '',
                                       @newFoodItemId = @foodItemId output;
SELECT @retval;
```

This is the output, letting us know that the transaction was still technically open and we could have committed any changes we wanted to:

```
----------------
In Error Handler

------------------
Savepoint Rollback

Msg 50000, Level 16, State 16, Procedure foodItem$insert, Line 57
Error occurred in procedure 'foodItem$insert', Original Message: 'The INSERT statement
conflicted with the CHECK constraint "CHKmenu_foodItem_description". The conflict occurred in
database "ContainedDatabase", table "menu.foodItem", column 'description'.'
```

You can see the constraint message, after the template error. Now, try to enter some data that is technically correct but is blocked by the trigger with the ROLLBACK:

```
DECLARE @foodItemId int, @retval int;
EXECUTE @retval = menu.foodItem$insert @name ='Big Burger',
                                       @description = 'Yummy Big Burger',
                                       @newFoodItemId = @foodItemId output;
SELECT @retval;
```

These results are a bit more mysterious, though the transaction is clearly in an error state. Since the rollback operation occurs in the trigger, once we reach the error handler, there is no need to do any savepoint or rollback, so it just finishes:

```
----------------
In Error Handler

Msg 50000, Level 16, State 16, Procedure foodItem$insert, Line 57
Error occurred in procedure 'foodItem$insert', Original Message: 'FoodItem's cannot be done
that way'
```

For the final demonstration, I will change the trigger to just do a RAISERROR, with no other error handling:

```
ALTER TRIGGER menu.foodItem$InsertTrigger
ON menu.foodItem
AFTER INSERT
AS
BEGIN
```


www.it-ebooks.info

```
   DECLARE @rowsAffected int, --stores the number of rows affected
           @msg varchar(2000); --used to hold the error message

   SET @rowsAffected = @@rowcount;

   --no need to continue on if no rows affected
   IF @rowsAffected = 0 RETURN;
   SET NOCOUNT ON; --to avoid the rowcount messages
   SET ROWCOUNT 0; --in case the client has modified the rowcount

  THROW 50000,'FoodItem''s cannot be done that way',16;
END;
```

Then, reexecute the previous statement that caused the trigger error:

```
----------------
In Error Handler
------------------
Transaction Doomed

Msg 50000, Level 16, State 16, Procedure foodItem$insert, Line 57
Error occurred in procedure 'foodItem$insert', Original Message: 'FoodItem's cannot be done that way'
```

Hence, our error handler covered all of the different bases of what can occur for errors. In each case, we got an error message that would let us know where an error was occurring and that it was an error. The main thing I wanted to show in this section is that error handling is messy and adding triggers, while useful, complicates the error handling process, so I would certainly use constraints as much as possible and triggers as rarely as possibly (the primary uses I have for them were outlined in Chapter 7). Also, be certain to develop an error handler in your T-SQL code and applications that is used with all of your code so that you capture all exceptions in a manner that is desirable for you and your developers.

Isolating Sessions

In the previous section, I introduced transactions, which are the foundation of the SQL Server concurrency controls. Even without concurrency they would be useful, but now, we are going to get a bit deeper into concurrency controls and start to demonstrate how multiple users can be manipulating and modifying the exact same data, making sure that all users get consistent usage of the data. Picture a farm with tractors and people picking vegetables. Both sets of farm users are necessary, but you definitely want to isolate their utilization from one another.

Of the ACID properties discussed earlier, isolation is probably the most difficult to understand and certainly the most important to get right. You probably don't want to make changes to a system and have them trampled by the next user any more than the farm hand wants to become tractor fodder (well, OK, perhaps his concern is a bit more physical—a wee bit at least). In this section, I will introduce a couple important concepts that are essential to building concurrent applications, both in understanding what is going on, and introduce the knobs you can use to tune the degree of isolation for sessions:

- **Locks***:* These are holds put by SQL Server on objects that are being used by users.
- **Isolation levels***:* These are settings used to control the length of time for which SQL Server holds onto the locks.

These two important things work together to allow you to control and optimize a server's concurrency, allowing users to work at the same time, on the same resources, while still maintaining consistency. However, just how consistent your data remains is the important thing, and that is what you will see in the "Isolation Levels" section.

www.it-ebooks.info

Locks

Locks are tokens laid down by the SQL Server processes to stake their claims to the different resources available, so as to prevent one process from stomping on another and causing inconsistencies or prevent another process from seeing data that has not yet been verified by constraints or triggers. They are a lot like the "diver down" markers that deep-sea divers place on top of the water when working below the water. They do this to alert other divers, pleasure boaters, fishermen, and others that they're below. Other divers are welcome, but a fishing boat with a trolling net should please stay away, thank you very much! Every SQL Server process applies a lock to anything it does to ensure that that other user processes know what they *are* doing as well as what they are *planning* to do and to ensure that other processes don't get in it's way. Minimally, a lock is always placed just to make sure that the database that is in use cannot be dropped.

The most common illustration of why locks are needed is called the *lost update*, as illustrated in Figure 11-1.

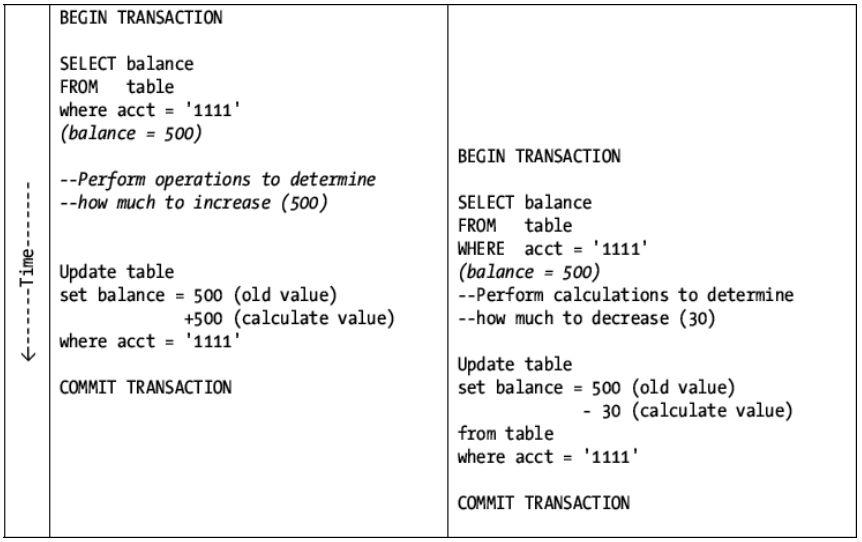


Figure 11-1. *A lost update illustration (probably one of the major inspirations for the other definition of multitasking: "screwing up everything simultaneously")*

In the scenario in Figure 11-1, you have two concurrent users. Each of these executes some SQL statements adding money to the balance, but in the end, the final value is going to be the wrong value, and 500 will be lost from the balance. Why? Because each user fetched a reality from the database that was correct at the time and then acted on it as if it would always be true.

Locks act as a message to other processes that a resource is being used, or at least probably being used. Think of a railroad-crossing sign. When the bar crosses the road, it acts as a lock to tell you not to drive across the tracks because the train is going to use the resource. Even if the train stops and never reaches the road, the bar comes down, and the lights flash. This lock can be ignored (as can SQL Server locks), but it's generally not advisable to do so, because if the train does come, you may not have the ability to go back to Disney World, except perhaps to the Haunted Mansion. (Ignoring locks isn't usually as messy as ignoring a train-crossing signal, unless you are creating the system that controls that warning signal. Ignore those locks—ouch.)

In this section, I will look at a few characteristics of locks:

- *Type of lock*: Indicates what is being locked
- *Mode of lock*: Indicates how strong the lock is

www.it-ebooks.info

Lock Types

If you've been around for a few versions of SQL Server, you probably know that since SQL Server 7.0, SQL Server primarily uses row-level locks. That is, a user locking some resource in SQL Server does it on individual rows of data, rather than on pages of data, or even on complete tables.

However, thinking that SQL Server only locks at the row level is misleading, as SQL Server can use six different types of locks to lock varying portions of the database, with the row being the finest type of lock, all the way up to a full database lock. And each of them will be used quite often. The types of locks in Table 11-1 are supported.

Table 11-1. *Lock Types*

Type of Lock	Granularity
Row or row identifier (RID)	A single row in a table
Key or key range	A single value or range of values (for example, to lock rows with values from A–M, even if no rows currently exist)
Page	An 8-KB index or data page
Extent	A group of eight 8-KB pages (64KB), generally only used when allocating new space to the database
HoBT	An entire heap or B-tree structure
Table	An entire table, including all rows and indexes
File	An entire file, as covered in Chapter 10
Application	A special type of lock that is user defined (will be covered in more detail later in this chapter)
Metadata	Metadata about the schema, such as catalog objects
Allocation unit	A group of 32 extents
Database	The entire database

■ **Tip** In terms of locks, database object locks (row, RID, key range, key, page, table, database) are all you have much knowledge of or control over in SQL Server, so these are all I'll cover. However, you should be aware that many more locks are in play, because SQL Server manages its hardware and internal needs as you execute queries. Hardware and internal resource locks are referred to as *latches*, and you'll occasionally see them referenced in SQL Server Books Online, though the documentation is not terribly deep regarding them. You have little control over them, because they control physical resources, like the lock on the lavatory door in an airplane. Like the lavatory, though, you generally only want one user accessing a physical resource at a time.

At the point of request, SQL Server determines approximately how many of the database resources (a table, a row, a key, a key range, and so on) are needed to satisfy the request. This is calculated on the basis of several factors, the specifics of which are unpublished. Some of these factors include the cost of acquiring the lock, the amount of resources needed, and how long the locks will be held (the next major section, "Isolation Levels," will

www.it-ebooks.info

discuss the factors surrounding the question "how long?"). It's also possible for the query processor to upgrade the lock from a more granular lock to a less specific type if the query is unexpectedly taking up large quantities of resources.

For example, if a large percentage of the rows in a table are locked with row locks, the query processor might switch to a table lock to finish out the process. Or, if you're adding large numbers of rows into a clustered table in sequential order, you might use a page lock on the new pages that are being added.

Lock Modes

Beyond the type of lock, the next concern is how strongly to lock the resource. For example, consider a construction site. Workers are generally allowed onto the site but not civilians who are not part of the process. Sometimes, however, one of the workers might need exclusive use of the site to do something that would be dangerous for other people to be around (like using explosives, for example.)

Where the type of lock defined the amount of the database to lock, the mode of the lock refers to how strict the lock is and how protective the engine is when dealing with other locks. Table 11-2 lists these available modes.

Table 11-2. *Lock Modes*

Mode	Description
Shared	This lock mode grants access for reads only. It's generally used when users are looking at but not editing the data. It's called "shared" because multiple processes can have a shared lock on the same resource, allowing read-only access to the resource. However, sharing resources prevents other processes from modifying the resource.
Exclusive	This mode gives exclusive access to a resource and can be used during modification of data also. Only one process may have an active exclusive lock on a resource.
Update	This mode is used to inform other processes that you're planning to modify the data but aren't quite ready to do so. Other connections may also issue shared, but not update or exclusive, locks while you're still preparing to do the modification. Update locks are used to prevent deadlocks (I'll cover them later in this section) by marking rows that a statement will possibly update, rather than upgrading directly from a shared lock to an exclusive one.
Intent	This mode communicates to other processes that taking one of the previously listed modes might be necessary. You might see this mode as intent shared, intent exclusive, or shared with intent exclusive.
Schema	This mode is used to lock the structure of an object when it's in use, so you cannot alter a table when a user is reading data from it.

Each of these modes, coupled with the granularity, describes a locking situation. For example, an exclusive table lock would mean that no other user can access any data in the table. An update table lock would say that other users could look at the data in the table, but any statement that might modify data in the table would have to wait until after this process has been completed.

To determine which mode of a lock is compatible with another mode of lock, we deal with lock compatibility. Each lock mode may or may not be compatible with the other lock mode on the same resource (or resource that contains other resources). If the types are compatible, two or more users may lock the same resource. Incompatible lock types would require the any additional users simply to wait until all of the incompatible locks have been released.

Table 11-3 shows which types are compatible with which others.

www.it-ebooks.info

Table 11-3. *Lock Compatibility Modes*

Mode	IS	S	U	IX	SIX	X
Intent shared (IS)	•	•	•	•	•	
Shared (S)	•	•	•			
Update (U)	•	•				
Intent exclusive (IX)	•			•		
Shared with intent exclusive (SIX)	•					
Exclusive (X)						

Although locks are great for data consistency, as far as concurrency is considered, locked resources stink. Whenever a resource is locked with an incompatible lock type and another process cannot use it to complete its processing, concurrency is lowered, because the process must wait for the other to complete before it can continue. This is generally referred to as *blocking*: one process is blocking another from doing something, so the blocked process must wait its turn, no matter how long it takes.

Simply put, locks allow consistent views of the data by only letting a single process modify a single resource at a time, while allowing multiple viewers simultaneous utilization in read-only access. Locks are a necessary part of SQL Server architecture, as is blocking to honor those locks when needed, to make sure one user doesn't trample on another's data, resulting in invalid data in some cases.

In the next section, I'll discuss isolation levels, which determine how long locks are held. Executing `SELECT * FROM sys.dm_os_waiting_tasks` gives you a list of all processes that tells you if any users are blocking and which user is doing the blocking. Executing `SELECT * FROM sys.dm_tran_locks` lets you see locks that are being held. SQL Server Management Studio has a decent Activity Monitor, accessible via the Object Explorer in the `Management` folder.

It's possible to instruct SQL Server to use a different type of lock than it might ordinarily choose by using *table hints* on your queries. For individual tables in a `FROM` clause, you can set the type of lock to be used for the single query like so:

```
FROM table1 [WITH] (<tableHintList>)
          JOIN table2 [WITH] (<tableHintList>)
```

Note that these hints work on all query types. In the case of locking, you can use quite a few. A partial list of the more common hints follows:

- `PageLock`: Forces the optimizer to choose page locks for the given table.
- `NoLock`: Leave no locks, and honor no locks for the given table.
- `RowLock`: Force row-level locks to be used for the table.
- `Tablock`: Go directly to table locks, rather than row or even page locks. This can speed some operations, but seriously lowers write concurrency.
- `TablockX`: This is the same as `Tablock`, but it always uses exclusive locks (whether it would have normally done so or not).
- `XLock`: Use exclusive locks.
- `UpdLock`: Use update locks.

Note that SQL Server can override your hints if necessary. For example, take the case where a query sets the table hint of `NoLock`, but then rows are modified in the table in the execution of the query. No shared locks are

www.it-ebooks.info

taken or honored, but exclusive locks are taken and held on the table for the rows that are modified, though not on rows that are only read (this is true even for resources that are read as part of a trigger or constraint).

A very important term that's you need to understand is "deadlock." A *deadlock* is a circumstance where two processes are trying to use the same objects, but neither will ever be able to complete because each is blocked by the other connection. For example, consider two processes (Processes 1 and 2), and two resources (Resources A and B). The following steps lead to a deadlock:

1. Process 1 takes a lock on Resource A, and at the same time, Process 2 takes a lock on Resource B.
2. Process 1 tries to get access to Resource B. Because it's locked by Process 2, Process 1 goes into a wait state.
3. Process 2 tries to get access to Resource A. Because it's locked by Process 1, Process 2 goes into a wait state.

At this point, there's no way to resolve this issue without ending one of the processes. SQL Server arbitrarily kills one of the processes, unless one of the processes has voluntarily raised the likelihood of being the killed process by setting `DEADLOCK_PRIORITY` to a lower value than the other. Values can be between integers –10 and 10, or `LOW` (equal to –5), `NORMAL` (0), or `HIGH` (5). SQL Server raises error 1205 to the client to tell the client that the process was stopped:

```
Server: Msg 1205, Level 13, State 1, Line 4
Transaction (Process ID 55) was deadlocked on lock resources with another process
and has been chosen as the deadlock victim. Rerun the transaction.
```

At this point, you could resubmit the request, as long as the call was coded such that the application knows when the transaction was started and what has occurred (something every application programmer ought to strive to do).

■ **Tip** Proper deadlock handling requires that you build your applications in such a way that you can easily tell how much of an operation succeeded or failed. This is done by proper use of transactions. A good practice is to send one transaction per batch from a client application. Keep in mind that the engine views nested transactions as one transaction, so what I mean here is to start and complete one high-level transaction per batch.

Deadlocks can be hard to diagnose, as you can deadlock on many things, even hardware access. A common trick to try to alleviate frequent deadlocks between pieces of code is to order object access in the same order in all code (so table `dbo.Apple`, `dbo.Bananna`, etc) if possible. This way, locks are more likely to be taken in the same order, causing the lock to block earlier, so that the next process is blocked instead of deadlocked.

An important consideration is that you really can't completely avoid deadlocks. Frequent deadlocks can be indicative of a problem with your code, but often, if you are running a very busy server, deadlocks happen, and the best thing to do is handle them by resubmitting the last transaction executed (too many applications just raise the deadlock as an error that users don't understand). Although frequent deadlocks are often an issue, it is very hard to code your system to be 100% safe from deadlocks (particularly when allowing users to share resources) so every call to the server should be aware that a deadlock could occur and, ideally, what to do with it.

Using SQL Server Profiler, you can add the `DeadLock Graph` event class to see deadlock events, which helps diagnose them. For more information about Profiler, check SQL Server Books Online.

www.it-ebooks.info

■ **Note** There's also a bulk update mode that I didn't mention; you use it to lock a table when inserting data in bulk into the table and applying the TABLOCK hint. It's analogous to an exclusive table lock for concurrency issues.

Isolation Levels

In the previous section, I said that locks are placed to make sure that a resource is protected while SQL Server is using it. But how long is the lock be held? Locks can be taken just for the amount of time it takes to get data from memory, or as long as a transaction is still open, even if that turns out to be hours. The *isolation level* is the setting that tells SQL Server how long to hold these locks, or even whether or not to take locks for read operations, and whether or not to honor other connections locks.

The safest method to provide consistency in operations would be to take an exclusive lock on the entire database, do your operations, and then release the lock. Then the next user does the same thing. Although this was somewhat common in early file-based systems, it isn't a reasonable alternative when you need to support 20,000 concurrent users (or even just a 10 data entry clerks, or perhaps automated users who do thousands of operations per second), no matter how beefy your hardware platform may be.

To improve concurrency, locks are held for the minimum time necessary to provide a reasonable amount of data consistency. (If the word "reasonable" concerns you, read on, because SQL Server defaults don't provide perfect coverage.) Isolation levels control how long locks are held, and there are five distinct levels (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE, and SNAPSHOT; each of which I will define in more detail later). From inside a transaction, locks can be held for a variable amount of time to protect the data that's being worked with. For example, consider the following hypothetical code snippet that illustrates an extremely typical mistake made by people just getting started(note that code like this in version 2008 or later should be migrated to use the new MERGE syntax, but this is going to remain a very common type of coding problem for years to come):

```
BEGIN TRANSACTION;
SAVE TRANSACTION savePoint;

IF EXISTS ( SELECT * FROM tableA WHERE tableAId = 'value' )
BEGIN
  UPDATE tableB
  SET status = 'UPDATED'
  WHERE tableAId = 'value';

  IF @@error <> 0
  BEGIN
    ROLLBACK TRANSACTION savePoint;
    THROW 50000, 'Error updating tableB',16;
  END;
END;
--usually followed by an insert
COMMIT TRANSACTION;
```

First, we check to see if a value exists in tableA. If it does, we update a value in tableB. On first glance, this seems safe—if a row exists when checked for in tableA, it will exist once the execution gets to the tableB update. However, how well this works is based solely on how long the locks are held on the SELECT from tableA, coupled with how long it takes to get to the UPDATE statement. Although the row might exist when the IF EXISTS block executed, what if a table lock exists on tableB when you try to execute the update of tableB, and the process gets blocked waiting for the lock to be cleared? While waiting for the table lock on tableB to be cleared, the key row that previously existed could have been deleted from tableA, if the lock isn't maintained on the row in tableA

www.it-ebooks.info

until the transaction is completed. You may think that this shouldn't be a problem, as perhaps the row was going to be deleted anyhow. But what if this row is important to a processing problem?

Your client believes that the row exists, because you updated it. And the delete operation may be removing the row because it was stale. In the section on optimistic locking techniques later in this chapter, I will demonstrate a way to make sure that the second process to try and update or delete has the most recent version when it completes, but in this case, unless you lock the row where tableAId = 'value' in the previous block of code's EXISTS criteria (IF EXISTS (SELECT * FROM tableA WHERE tableAId = 'value')) the delete may happen even before the UPDATE, depending on what process first gets the query processor when the lock occurs.

What's the major problem here, and why is it usually a major problem? Under the default isolation level in which SQL Server connections operate, no lock would have been kept on tableA, leaving a potential hole in your data integrity if another user makes a change to the table before your transaction is complete. Usually, however, if you have checks on tableA that validate the effects on tableB's integrity, the locks from the modification operations will protect you from integrity issues.

Deeply ingrained in the concepts of isolation levels are the concepts of *repeatable reads* and *phantom rows*. Consider that you execute a statement such as the following within a transaction. Logically, it seems like you should get back exactly the same data, but this is not always the case.

```
BEGIN TRANSACTION;
SELECT * FROM table;
```

And the following rows were returned:

```
ColumnName
-----------
row1
row2
```

For this SELECT statement to claim to support repeatable reads within a transaction, you must be able to execute it multiple times in the transaction and get back at least the same results, possibly more. This means that no other user could change the data that had been retrieved in the operation. Other users are allowed to create new rows, so on a subsequent query to the table, you might get back the following results:

```
ColumnName
-----------
row1
row2
row3
```

Note that the term "repeatable read" can seem confusing (it does to me, but I had no say in the matter!) because the exact results of the read weren't repeatable, but that's how it's defined. The value row3 is called a phantom row, because it just appears out of nowhere whenever you execute the query a second time.

The following bulleted list contains the isolation levels to adjust how long locks are held to prevent phantom rows and nonrepeatable reads:

- READ UNCOMMITTED: Doesn't honor or take locks, unless data is modified.
- READ COMMITTED: Takes and honors locks, but releases read locks after data is retrieved. Allows phantom rows and nonrepeatable reads.

www.it-ebooks.info

- REPEATABLE READ: Holds locks for the duration of the transaction to prevent users from changing data. Disallows nonrepeatable reads but allows phantom rows.
- SERIALIZABLE: Like REPEATABLE READ, but adds locks on ranges of data to make sure no new data is added. Holds these locks until the transaction is completed. Disallows phantom rows and nonrepeatable reads.
- SNAPSHOT: Allows the user to look at data as it was when the transaction started (existed as of SQL Server 2005).

The syntax for setting the isolation level is as follows:

```
SET TRANSACTION ISOLATION LEVEL <level>;
```

<level> is any of the five preceding settings. The default isolation level is READ COMMITTED and is a good balance between concurrency and integrity. It does bear mentioning that READ COMMITTED isn't always the proper setting. Quite often, when only reading data, the SNAPSHOT isolation level gives the best results, though not properly setting up your servers can have some serious performance implications (more on the reasons for that in the section dedicated to SNAPSHOT).

Referring to the previous example code block—checking that a value exists in one table, then modifying another—keep in mind that the types of tables that tableA and tableB represent will greatly affect the need to change the isolation level. In that case, using the REPEATABLE READ isolation level would suffice, because you are looking for the case where the row existed. REPEATABLE READ will allow phantoms, but if one row exists and you add another, existence is still guaranteed if another row is created.

Keep in mind that locks aren't just held for operations that you directly execute. They can be held for any constraints that fire to check existence in other tables and any code executed in trigger code. The isolation level in effect also controls how long these locks are held. Understanding that fact alone will make you a much better performance tuner, because you won't just look on the surface but will know to dig deep into the code to figure out what is going on.

When considering solutions, you must keep in mind locking and isolation levels. As more and more critical solutions are being built on SQL Server, it's imperative to make absolutely sure to protect data at a level that's commensurate with the value of the data. If you are building procedures to support a system on a space shuttle or a life support system, this becomes more important than it would be in the case of a sales system, a pediatrician's schedule, or like we set up in Chapter 6, a simple messaging system.

In some cases, losing some data really doesn't matter. It is up to you when you are designing your system to truly understand that particular system's needs. In the "Coding for Integrity and Concurrency" section, I'll look at coding schemes aimed at improving the concurrency of your stored procedure programs.

■ **Tip** The IF EXISTS() THEN . . . ELSE . . . scenario mentioned earlier cannot be managed simply with isolation levels. In the next section, when I discuss pessimistic locking, I will present a solution using application locks that can be fitted to perform the "perfect" single threading solution.

In the next subsections, I'll briefly discuss the different isolation levels and demonstrate how they work using the following table. After the specific isolation levels, I will have a section that covers a database setting that alters how the isolation levels work called READ COMMITTED SNAPSHOT as well. (Again, build these in any database that you choose. I'll create them in tempdb.)

```
CREATE TABLE dbo.testIsolationLevel
(
  testIsolationLevelId int NOT NULL IDENTITY(1,1)
```


www.it-ebooks.info

```
              CONSTRAINT PKtestIsolationLevel PRIMARY KEY,
  value varchar(10) NOT NULL
);

INSERT dbo.testIsolationLevel(value)
VALUES ('Value1'),
       ('Value2');
```

■ **Tip** Just as for locking modes, there are table query hints to apply an isolation level only to a given table in a query, rather than an entire query. These hints are `READUNCOMMITTED`, `READCOMMITTED`, `REPEATABLEREAD`, `SNAPSHOT`, and `SERIALIZABLE`, and they behave as their corresponding isolation levels do, only with respect to a single table in a query.

When you are coding or testing, checking to see what isolation level you are currently executing under can be useful. To do this, you can look at the results from `sys.dm_exec_sessions`:

```
SELECT CASE transaction_isolation_level
            WHEN 1 THEN 'Read Uncomitted'          WHEN 2 THEN 'Read Committed'
            WHEN 3 THEN 'Repeatable Read'          WHEN 4 THEN 'Serializable'
            WHEN 5 THEN 'Snapshot'                 ELSE 'Unspecified'
       END
FROM  sys.dm_exec_sessions
WHERE session_id = @@spid;
```

Unless you have already changed it, the default (and what you should get from executing this query in your connection) is `Read Committed`. Change the isolation level to serializable like so:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Then, reexecute the query, and the results will now show that the isolation level is currently serializable. In the following sections, I will show you why you would want to change the isolation level at all.

■ **Tip** I have included all of the code for these chapters in a single file, but you will want to start your own connections for `CONNECTION A` and `CONNECTION B`. All of the example code requires multiple connections to execute, in order to allow for concurrency.

READ UNCOMMITTED

Ignore all locks, and don't issue locks. Queries can see any data that has been saved to the table, regardless of whether or not it's part of a transaction that hasn't been committed (hence the name). However, `READ UNCOMMITTED` still leaves exclusive locks if you do modify data, to keep other users from changing data that you haven't committed.

For the most part, `READ UNCOMMITTED` is a good tool for developers to use to check the progress of operations and to look at production systems when `SNAPSHOT` isn't available. It should not be used as a performance tuning tool, however, because all of your code should use only committed trustable data that has passed the

www.it-ebooks.info

requirements of the constraints and triggers you have implemented. For example, say you execute the following code on one connection:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; --this is the default, just
                                                --setting for emphasis
BEGIN TRANSACTION
INSERT INTO dbo.testIsolationLevel(value);
VALUES('Value3');
```

Then, you execute on a second connection:

```
--CONNECTION B
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT *
FROM dbo.testIsolationLevel;
```

This returns the following results:

```
testIsolationLevelId    value
--------------------    ------
1                       Value1
2                       Value2
3                       Value3
```

Being able to see locked data is quite valuable, especially when you're in the middle of a long-running process. That's because you won't block the process that's running, but you can see the data being modified. There is no guarantee that the data you see will be correct (it might fail checks and be rolled back), but for looking around and some reporting needs, this data might be good enough.

Finally, commit the transaction you started earlier:

```
--CONNECTION A

COMMIT TRANSACTION;
```

■ **Caution** Ignoring locks using `READ UNCOMMITTED` is almost never a good way to build highly concurrent database systems! Yes, it is possible to make your applications screamingly fast, because they never have to wait for other processes. There is a reason for this waiting. Consistency of the data you read is highly important and should not be taken lightly. Using `SNAPSHOT` or `READ COMMITTED SNAPSHOT`, which I will cover later in the chapter, will give you sort of the same concurrency without reading dirty data.

READ COMMITTED

`READ COMMITTED` is the default isolation level as far as SQL Server is concerned, and as the name states, it prevents you from seeing uncommitted data. Be careful that your toolset may or may not use it as its default (some toolsets use `SERIALIZABLE` as the default, which, as you will see is pretty tight and is not great for concurrency). All shared and update locks are released as soon as the process is finished using the resource. Exclusive locks are held

www.it-ebooks.info

until the end of the transaction. Data modifications are usually executed under this isolation level. However, understand that this isolation level isn't perfect, as there isn't protection for repeatable reads or phantom rows. This means that as the length of the transaction increases, there's a growing possibility that some data that was read during the first operations within a transaction might have been changed or deleted by the end of the transaction. It happens extremely rarely when transactions are kept short, so it's generally considered an acceptable risk—for example:

```
--CONNECTION A

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;
SELECT * FROM dbo.testIsolationLevel;
```

You see all the rows in the table from the previous section (though the `testIsolationLevelId` might be different if you had errors when you built your code). Then, on the second connection, delete a row:

```
--CONNECTION B
DELETE FROM dbo.testIsolationLevel
WHERE testIsolationLevelId = 1;
```

Finally, go back to the other connection and execute, still within the transaction:

```
--CONNECTION A
SELECT *
FROM dbo.testIsolationLevel;
COMMIT TRANSACTION;
```

This returns the following results:

```
testIsolationLevelId    value
--------------------    ------
2                       Value2
3                       Value3
```

The first time you grasp this topic well (hopefully now, but it may take a bit of time for it to sink in), you may very well panic that your data integrity is in trouble. You are right in one respect. There are some holes in the default isolation level. However, since most referential integrity checks are done based on the existence of some data, the impact of `READ COMMITTED` is lessened by the fact that most operations in an OLTP database system are inserts and updates. The impact is further lessened because relationships are pretty well guarded by the fact that deleting the parent or child row in a relationship requires a lock on the other rows. So if someone tries to modify the parent and someone else tries to modify the child, one process will be blocked by the other.

Beyond the fact that relationships require locked checks in `READ COMMITTED` isolation, the key to the success of using this isolation level is simple probability. The chances of two users stepping on each other's processes within milliseconds is pretty unlikely, even less likely is the scenario that one user will do the exact thing that would cause inconsistency. However, the longer your transactions and the higher the concurrent number of users on the system, the more likely that `READ COMMITTED` will produce anomalies.

In my 18 years of using SQL Server, the primary issues I have found with `READ COMMITTED` have centered exclusively on checking/retrieving a value and then going back later and using that value. If you do much of that, and it is important that the situation remain the same until you use the value, consider implementing your code using a higher level of isolation.

www.it-ebooks.info

For example, consider the issues involved in implementing a system to track drugs given to a patient in a hospital. For a system such as this, you'd never want to give a user too much medicine accidentally because when you started a process to set up a schedule via a batch system, a nurse was administering the dosage off schedule. Although this situation is unlikely, as you will see in the next few sections, an adjustment in isolation level would prevent it from occurring at all.

REPEATABLE READ

The `REPEATABLE READ` isolation level includes protection from data being deleted from under your operation. Shared locks are now held during the entire transaction to prevent other users from modifying the data that has been read. You would be most likely to use this isolation level if your concern is the absolute guarantee of existence of some data when you finish your operation.

As an example on one connection, execute the following statement:

```
--CONNECTION A

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRANSACTION;
SELECT * FROM dbo.testIsolationLevel;
```

This returns the following:

```
testIsolationLevelId    value
--------------------    ------
2                       Value2
3                       Value3
```

Then, on a different connection, run the following:

```
--CONNECTION B

INSERT INTO dbo.testIsolationLevel(value)
VALUES ('Value4');
```

This executes, but try executing the following code:

```
--CONNECTION B
DELETE FROM dbo.testIsolationLevel
WHERE value = 'Value3';
```

You go into a blocked state: `CONNECTION B` will need an exclusive lock on that particular value, because deleting that value would cause the results from `CONNECTION A` to return fewer rows. Back on the first connection, run the following code:

```
--CONNECTION A

SELECT * FROM dbo.testIsolationLevel;
COMMIT TRANSACTION;
```

This will return the following:

www.it-ebooks.info

```
testIsolationLevelId    value
--------------------    ------
2                       Value2
3                       Value3
4                       Value4
```

And immediately, the batch on the other connection will complete. Now, view the data (from either connection):

```
--CONNECTION A
SELECT * FROM dbo.testIsolationLevel;
```

This returns

```
testIsolationLevelId    value
--------------------    ------
2                       Value2
4                       Value4
```

The fact that other users may be changing the data you have locked can be a very serious concern for the perceived integrity of the data. If the user on connection A goes right back and the row is deleted, that user will be confused. Of course, nothing can really be done to solve this problem, as it is just a fact of life. In the "Optimistic Locking" section, I will present a method of making sure that one user doesn't crush the changes of another; the method could be extended to viewed data, but generally, this is not the case for performance reasons.

In the end, you can implement most any scheme to protect the data, but all you are doing is widening the window of time where users are protected. No matter what, once the user relinquishes transactional control on a row, it will be fair game to other users without some form of workflow system in place (a topic that is well beyond the scope of my book, though once you are finished reading this book, you could design and create one!).

SERIALIZABLE

`SERIALIZABLE` takes everything from `REPEATABLE READ` and adds in phantom-row protection. SQL Server accomplishes this by taking locks not only on existing data that it has read but on any ranges of data that *could* match any SQL statement executed. This is the most restrictive isolation level and is the best in any case where data integrity is absolutely necessary. It can cause lots of blocking; for example, consider what would happen if you executed the following query under the `SERIALIZABLE` isolation level:

```
SELECT *
FROM dbo.testIsolationLevel;
```

No other user will be able to modify the table until all rows have been returned and the transaction it was executing within (implicit or explicit) is completed.

■ **Note** Be careful. I said, "No other user will be able to modify the table . . . ." I didn't say "read." Readers leave shared locks, not exclusive ones. This caveat is something that can be confusing at times when you are trying to write safe but concurrent SQL code. Other users could fetch the rows into cache, make some changes in memory,

www.it-ebooks.info

and then write them later. We will look at techniques to avoid this issue later in this chapter as we discuss optimistic locking techniques.

If lots of users are viewing data in the table under any of the previously mentioned isolation levels, it can be difficult to get any modifications done. If you're going to use SERIALIZABLE, you need to be careful with your code and make sure it only uses the minimum number of rows needed (especially if you are not using SNAPSHOT isolation level for read processes, as covered in the next section).

Execute this statement on a connection to simulate a user with a table locked:

```
--CONNECTION A

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;
SELECT * FROM dbo.testIsolationLevel;
```

Then, try to add a new row to the table:

```
--CONNECTION B

INSERT INTO dbo.testIsolationLevel(value)
VALUES ('Value5');
```

Your insert is blocked. Commit the transaction on the A connection:

```
--CONNECTION A

SELECT * FROM dbo.testIsolationLevel;
COMMIT TRANSACTION;
```

This returns the following:

```
testIsolationLevelId    value
--------------------    ------
2                       Value2
4                       Value4
```

The results are the same. However, this unblocks CONNECTION B, and running the SELECT again, you will see that the contents of the table are now the following:

```
testIsolationLevelId    value
--------------------    ------
2                       Value2
4                       Value4
5                       Value5
```

It is important to be careful with the SERIALIZABLE isolation level. I can't stress enough that multiple readers can read the same data, but no one can update it while others are reading. Too often, people take this to mean that they can read some data and be guaranteed that no other user might have read it also, leading occasionally to inconsistent results and more frequently to deadlocking issues.

www.it-ebooks.info

SNAPSHOT

SNAPSHOT isolation was one of the major cool new features in SQL Server 2005, and it continues to be one of my favorites (particularly the READ COMMITTED SNAPSHOT variant that is mentioned later in this section). It lets you read the data as it was when the transaction started, regardless of any changes. It's a special case, because although it doesn't technically allow for phantom rows, nonrepeatable reads, or dirty reads from any queries within the transaction, it doesn't necessarily represent the current state of the data. You might check a value in a table at the beginning of the transaction and it's in the physical table, but later, you requery the table and it is no longer there. As long as you are inside the same transaction, even though the value exists in your virtual table, it needn't exist in the physical table any longer (in fact, the physical table needn't exist either!). This provides that the results of your query will reflect a consistent state of the database at some time, which is generally very desirable.

What makes SNAPSHOT particularly useful is that it doesn't use locks in the normal way, because it looks at the data as it was at the start of the transaction. Modifying data under this isolation level has its share of problems, which I'll demonstrate later in this section. However, I don't want completely to scare you off, as this isolation level can become a major part of a highly concurrent design strategy (particularly useful for reads in an optimistic locking strategy, which the last sections of this chapter cover).

The largest downside is the effect it can have on performance if you are not prepared for it. This history data is written not only to the log, but the data that will be used to support other users that are in a SNAPSHOT isolation level transaction is written to the tempdb. Hence, if this server is going to be very active, you have to make sure that tempdb is up to the challenge, especially if you're supporting large numbers of concurrent users.

The good news is that, if you employ the strategy of having readers use SNAPSHOT isolation level , data readers will no longer block data writers (in any of the other isolation levels), and they will always get a transactionally consistent view of the data. So when the vice president of the company decides to write a 20-table join query in the middle of the busiest part of the day, all other users won't get stuck behind him with data locks. The better news is that he won't see the mistaken ten-million-dollar entry that one of the data-entry clerks added to the data that the check constraint hasn't had time to deny yet (the vice president would have seen the error if you were using the READ UNCOMMITTED solution, which is the unfortunate choice of many novice performance tuners). The bad news is that eventually the vice president's query might take up all the resources and cause a major system slowdown that way. (Hey, if it was too easy, companies wouldn't need DBAs. And I, for one, wouldn't survive in a nontechnical field.)

To use (and demonstrate) SNAPSHOT isolation level, you have to alter the database you're working with (you can even do this to tempdb):

```
ALTER DATABASE tempDb
     SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Now, the SNAPSHOT isolation level is available for queries.

■ **Caution** The SNAPSHOT isolation level uses copies of affected data placed into tempdb. Because of this, you should make sure that your tempdb is set up optimally.

Let's look at an example. On the first connection, start a transaction and select from the -testIsolationLevel table:

```
--CONNECTION A

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```


www.it-ebooks.info

```
BEGIN TRANSACTION;
SELECT * FROM dbo.testIsolationLevel;
```

This returns the following results:

```
testIsolationLevelId    value
--------------------    ------
2                       Value2
4                       Value4
5                       Value5
```

On a second connection, run the following:

```
--CONNECTION B

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO dbo.testIsolationLevel(value)
VALUES ('Value6');
```

This executes with no waiting. Going back to Connection A, reexecuting the SELECT returns the same set as before, so the results remain consistent. On Connection B, run the following DELETE statement:

```
--CONNECTION B

DELETE FROM dbo.testIsolationLevel
WHERE  value = 'Value4';
```

This doesn't have to wait either. Going back to the other connection again, nothing has changed.

```
--CONNECTION A

SELECT * FROM dbo.testIsolationLevel;
```

This still returns

```
testIsolationLevelId value
-------------------- ------
2                    Value2
4                    Value4
5                    Value5
```

So what about modifying data in SNAPSHOT isolation level? If no one else has modified the row, you can make any change:

```
--CONNECTION A

UPDATE dbo.testIsolationLevel
SET    value = 'Value2-mod'
WHERE  testIsolationLevelId = 2;
```


www.it-ebooks.info

This runs, but going back to the B connection, if you try to select this row

```
--CONNECTION B
SELECT * FROM dbo.testIsolationLevel
```

you will find the query is blocked, and the connection is forced to wait, because this row is new and has an exclusive lock on it, and connection B is not in SNAPSHOT ISOLATION level.

Commit the transaction in CONNECTION A, and you'll see rows such as these:

```
--CONNECTION A

COMMIT TRANSACTION;
SELECT * FROM dbo.testIsolationLevel;
```

This returns the current contents of the table:

```
testIsolationLevelId    value
--------------------    ----------
2                       Value2-mod
5                       Value5
6                       Value6
```

The messy and troubling bit with modifying data under the SNAPSHOT isolation level is what happens when one user modifies a row that another user has also modified and committed the transaction for. To see this, in CONNECTION A run the following, simulating a user fetching some data into the cache:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;

--touch the data
SELECT * FROM dbo.testIsolationLevel;
```

This returns the same results as just shown. Then, a second user changes the value:

```
--CONNECTION B
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; --any will do

UPDATE dbo.testIsolationLevel
SET    value = 'Value5-mod'
WHERE  testIsolationLevelId = 5; --might be different surrogate key value in yours
```

Next, the user on CONNECTION A tries to update the row also:

```
--CONNECTION A

UPDATE dbo.testIsolationLevel
SET    value = 'Value5-mod'
WHERE  testIsolationLevelId = 5; --might be different in yours
```

As this row has been deleted by a different connection, the following error message rears its ugly head:

```
Msg 3960, Level 16, State 2, Line 2
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot
isolation to access table 'dbo.testIsolationLevel' directly or indirectly in database 'tempdb'
```


www.it-ebooks.info

```
to update, delete, or insert the row that has been modified or deleted by another transaction.
Retry the transaction or change the isolation level for the update/delete statement.
```

As such, strictly for simplicity's sake, I recommend that almost all retrieval-only operations can execute under the SNAPSHOT isolation level, and the procedures that do data modifications execute under the READ COMMITTED isolation level. As long as data is only read, the connection will see the state of the database as it was when the data was first read.

A strong word of caution—if you do data validations under SNAPSHOT isolation level and you do data-checking logic such as in a trigger or procedure, the data might already be invalid in the live database, especially if the transaction runs long. This invalid data is far worse than REPEATABLE READ, where the data is always valid when the check is done but might be changed after the violation. However, you should note that FOREIGN KEY constraints, when doing a modification, are smart enough to use the same sorts of locks as READ COMMITTED would to protect against this sort of issue. My suggestion (if you ignore the suggestion not to do writes under SNAPSHOT) would be to manually code SET TRANSACTION ISOLATION LEVEL READ COMMITTED or REPEATABLE READ or SERIALIZABLE where necessary in your modification procedures or triggers to avoid this sort of issue.

READ COMMITTED SNAPSHOT (Database Setting)

The database setting READ_COMMITTED_SNAPSHOT changes the isolation level of READ COMMITTED to behave very much like SNAPSHOT isolation level on a statement level.

Note that I said "statement" and not "transaction." In SNAPSHOT isolation level, once you start a transaction, you get a consistent view of the database *as it was* when the transaction started until you close it. READ_COMMITTED_SNAPSHOT gives you a consistent view of the database for a single statement. Set the database into this mode as follows:

```
--must be no active connections other than the connection executing
--this ALTER command Does not require ALLOW_SNAPSHOT_ISOLATION enabled.
ALTER DATABASE <databasename>
    SET READ_COMMITTED_SNAPSHOT ON;
```

When you do this, every *statement* is now in SNAPSHOT isolation level by default. For example, imagine you're at the midpoint of the following pseudo-batch:

```
BEGIN TRANSACTION;
SELECT column FROM table1;
--midpoint
SELECT column FROM table1;
COMMIT TRANSACTION;
```

If you're in SNAPSHOT isolation level, table1 could change completely—even get dropped—and you wouldn't be able to tell when you execute the second SELECT statement. You're given a consistent view of the database for reading. With the READ_COMMITTED_SNAPSHOT database setting turned on, in a READ COMMITTED isolation level transaction, your view of table1 would be consistent with how it looked when you started reading, but when you started the second pass through the table, it might not match the data the first time you read through. This behavior is similar to plain READ COMMITTED, except that you don't see any new phantoms or nonrepeatable reads while retrieving rows produced during the individual statement (other users can delete and add rows while you scan through the table, but you won't be affected by the changes), and SQL Server doesn't need to take locks or block other users.

Using READ_COMMITTED_SNAPSHOT can actually perform tremendously better than just using READ COMMITTED, though it does suffer from the same (or maybe worse) issues with data. You should remember that in the previous section on READ COMMITTED, I noted that because SQL Server releases locks immediately after reading the data,

www.it-ebooks.info

another user could come behind you and change data that you just finished using to do some validation. This same thing is true for READ_COMMITTED_SNAPSHOT, but the window of time can be slightly longer because SQL Server reads only history as it passes through different tables. This amount of time is generally insignificant and usually isn't anything to worry about, but it *can* be important based on the type of system you're creating.

For places where you might need more safety, consider using the higher isolation levels, such as REPEATABLE READ or SERIALIZABLE. I would certainly suggest that, in the triggers and modification procedures that you build using this isolation level, you consider the upgraded isolation level. The best part is that basic readers who just want to see data for a query or report will not be affected. Later in this chapter, when I present the mechanism for optimistic locking, you will see that whether or not a reading user gets an old version doesn't really matter. Users will never be allowed to modify anything but the rows that look exactly like the ones that they fetched.

SNAPSHOT isolation level and the READ_COMMITTED_SNAPSHOT settings are very important aspects of SQL Server's concurrency feature set. They cut down on blocking and the need to use the dirty reads to look at active OLTP data for small reports and for read-only queries to cache data for user-interface processes.

■ **Note** READ COMMITTED SNAPSHOT is the feature that saved one of the major projects I worked on after version 2005 was released. We tried and tried to optimize the system under basic READ COMMITTED, but it was not possible, mostly because we had no control over the API building the queries that were used to access the database.

Coding for Integrity and Concurrency

When building database systems, you must consider that multiple users will be attempting to modify your data at the same time. So far in this chapter, I've talked at length about the different mechanisms, such as transactions, isolation levels, and so on, for protecting your data. Now, I'll present some of the coding mechanisms that keep your users from stepping on one another.

The general progression of events for most applications is the same: fetch some data for a user or a process to look at, operate on this data, make changes to the data, or make some decision based on the retrieved values. Once the users have performed their operations, they'll either commit their changes to the database or possibly save data to a different table based on their decisions.

Our coding decisions generally surround how to deal with the lag time while the users have the data cached on their clients. For example, what happens if a different user wants to access and make a change to the same data?

For this situation, you can use a couple common schemes while coding your database application:

- **Pessimistic locking***:* Assume it's likely that users will try to modify the same data, so single-thread access to important resources.
- **Optimistic locking***:* Assume it's unlikely that users will try to modify the exact same row at the same time other users want to. Only verify that the cached data is valid when a user wants to change the data.

Using one or parts of both of these schemes, it's usually possible to protect data in a multiuser system at an acceptable level of integrity and concurrency.

Pessimistic Locking

A pessimistic locking scheme is restrictive. Generally, the idea is straightforward: put a lock on a resource, keeping all others out; do your work with the resource; then, release the lock. The goal is to eliminate any chance of contention. In SQL code, the process is a bit difficult to get right, as you will see, but the basics are as follows:

www.it-ebooks.info

begin a transaction, most likely a serializable one; fetch the data; manipulate the data; modify the data; and finally, commit the transaction. The goal is to serialize or single-thread all access to the resource in which the process is interested, making sure that no other user can modify or even view the data being worked on.

The most significant concern is blocking all access to given resources. This sounds easy and reasonable, but the main issue is that any query to a locked resource has to wait for the user to complete access. Even if the parts of the resource won't be involved in the answer to a query, if a locked resource *might* be involved, unnecessary blocking may occur.

For example, say one user has a single row locked in a table. The next user executes a different query that requires a table scan on the same table. Even if the results of this query needn't use the locked row, the second user will be blocked until the other connection has completed, because SQL Server won't know if the next user needs the row until it's unlocked.

■ **Note** You might be thinking that SQL Server could simply check to see if the locked resource would be needed. However, this cannot be known, because once a row is locked with a noncompatible lock, all other users must assume that the values might change. Hence, you're forced to wait until the lock is dropped.

Any users who need data that this next user has locked also have to wait, and soon, a chain of users is waiting on one particular user. Except for one thing—time—all this might even be reasonable. If the lock only lasted milliseconds (or possibly seconds), this could be fine for many systems. Small applications based on file managers have implemented concurrency this way for years. However, what if the user decides to take a break? (This can be a common issue with smaller systems.) All other users have to wait until this user finishes accessing the data, and if this user has modified one piece of data (possibly with complex triggers) and still has more to go, access might be blocked to most of the system data because that user was forgetful and didn't click the Save button.

It's possible to relieve some of the long-term stress on the system by reducing the amount of time for which locks can be held, such as setting time limits on how long the user can keep the data before rolling back the transaction. However, either way, it's necessary to block access to large quantities of data for a more-than-reasonable period of time, because you'd need to lock any domain tables that the users will rely on to choose values for their tables so they change no related table values or any related data that other users might need.

Implementing pessimistic locks isn't all that easy, because you have to go out of your way to force locks on data that keep other users from even viewing the data. One method is to lock data using exclusive lock hints, coupled with the `SERIALIZABLE` isolation level, when you fetch data and maintain the connection to the server as you modify the data. This approach is messy and will likely cause lots of undesired locks if you aren't extremely careful how you write queries to minimize locking.

■ **Caution** If the page has not been dirtied, even if an exclusive lock exists on the row or page, another reader can get access to the row for viewing. You need to actually modify (even to the same value) the row to dirty the page if you want to hold the lock, or use a `PAGELOCK` and `XLOCK` hint (see Microsoft Knowledgebase Article 324417), though doing so will lock the entire page.

SQL Server does have a built-in method you can use to implement a form of pessimistic locking: SQL Server *application locks*. These locks, just like other locks, must be taken inside a transaction (executing the procedures

www.it-ebooks.info

without a transaction will get you a nasty error message). The real downside is that enforcement and compliance are completely optional. If you write code that doesn't follow the rules and use the proper application lock, you will get no error letting you know. The commands that you have to work with application locks are as follows:

- `sp_getAppLock`: Use this to place a lock on an application resource. The programmer names application resources, which can be named with any string value. In the string, you could name single values or even a range.
- `sp_releaseAppLock`: Use this to release locks taken inside a transaction.
- `APPLOCK_MODE`: Use it to check the mode of the application lock.
- `APPLOCK_TEST`: Use this one to see if you could take an application lock before starting the lock and getting blocked.

As an example, we'll run the following code. We'll implement this on a resource named `'invoiceId=1'`, which represents an invoice that we'll lock. We'll set it as an exclusive lock so no other user can touch it. In one connection, we run the following code:

```
--CONNECTION A

BEGIN TRANSACTION;
    DECLARE @result int
    EXEC @result = sp_getapplock @Resource = 'invoiceId=1', @LockMode = 'Exclusive';
    SELECT @result;
```

This returns `0`, stating that the lock was taken successfully. Now, if another user tries to execute the same code to take the same lock, the second process has to wait until the first user has finished with the resource `'invoiceId=1'`:

```
--CONNECTION B

BEGIN TRANSACTION;
    DECLARE @result int;
    EXEC @result = sp_getapplock @Resource = 'invoiceId=1', @LockMode = 'Exclusive';
    SELECT @result;
```

This transaction has to wait. Let's cancel the execution, and then execute the following code using the `APPLOCK_TEST` function (which has to be executed in a transaction context) to see if we can take the lock (allowing the application to check before taking the lock):

```
--CONNECTION B

BEGIN TRANSACTION
SELECT APPLOCK_TEST('public','invoiceId=1','Exclusive','Transaction')
                                                    as CanTakeLock
ROLLBACK TRANSACTION
```

This returns `0`, meaning we cannot take this lock currently. `APPLOCK`s can be a great resource for building locks that are needed to implement locks that are "larger" than just SQL Server objects. The key is that every user that is to participate in the `APPLOCK` must implement the locking mechanism, and every user must honor the locks taken. In the next section, I will show you a very common and useful technique using `APPLOCK`s to create a pessimistic lock based on the application lock to single thread access to a given block of code.

www.it-ebooks.info

■ **Tip** You can use application locks to implement more than just pessimistic locks using different lock modes other than exclusive, but exclusive is the mode you'd use to implement a pessimistic locking mechanism. For more information about application locks, SQL Server Books Online gives some good examples and a full reference to using application locks.

Implementing a Single-Threaded Code Block

The problem of the critical section is a very common problem. Very often, it is troublesome for more than one connection to have access to a given section of code. For example, you might need to fetch a value, increment it, and keep the result unique among other callers that could be calling simultaneously.

The general solution to the single threading problem is to exclusively lock the resources that you need to be able to work with, forcing all other users to wait even for reading. In some cases, this technique will work great, but it can be troublesome in cases like the following:

- The code is part of a larger set of code that may have other code locked in a transaction, blocking users' access to more than you expect. You are allowed to release the application lock in the transaction to allow other callers to continue.
- Only one minor section of code needs to be single threaded, and you can allow simultaneous access otherwise.
- The speed in which the data is accessed is so fast that two processes are likely to fetch the same data within microseconds of each other.
- The single threading is not for table access. For example, you may want to write to a file of some sort or use some other resource that is not table based.

The following technique will leave the tables unlocked while manually single threading access to a code block (in this case, getting and setting a value), using an application lock to lock a section of code.

■ **Note** An application lock must be used and honored manually in every piece of code where the need to lock the data matters, so there is a loss of safety associated with using application locks rather than data-oriented locks. If there is any concern with what other processes might do, be sure to still assign proper concurrency and locking hints to that code also.

To demonstrate a very common problem of building a unique value without using identities (for example, if you have to create an account number with special formatting/processing), I have created the following table:

```
CREATE TABLE applock
(
    applockId int NOT NULL CONSTRAINT PKapplock PRIMARY KEY,
                              --the value that we will be generating
                              --with the procedure
    connectionId int NOT NULL,  --holds the spid of the connection so you can
                              --who creates the row
    insertTime datetime2(3) NOT NULL DEFAULT (SYSDATETIME()) --the time the row was created, so
                                                             --you can see the progression
```


www.it-ebooks.info

```
);
```

Next, a procedure that starts an application lock fetches some data from the table, increments the value, and stores it in a variable. I added a delay parameter, so you can tune up the problems by making the delay between incrementing and inserting more pronounced. There is also a parameter to turn on and off the application lock (noted as `@useApplockFlag` in the parameters), and that parameter will help you test to see how it behaves with and without the application lock.

```
CREATE PROCEDURE applock$test
(
      @connectionId int,
      @useApplockFlag bit = 1,
      @stepDelay varchar(10) = '00:00:00'
) as
SET NOCOUNT ON
BEGIN TRY
    BEGIN TRANSACTION
        DECLARE @retval int = 1;
        IF @useApplockFlag = 1 --turns on and off the applock for testing
            BEGIN
                EXEC @retval = sp_getapplock @Resource = 'applock$test',
                                                @LockMode = 'exclusive';
                IF @retval < 0
                    BEGIN
                        DECLARE @errorMessage nvarchar(200);
                        SET @errorMessage = CASE @retval
                                    WHEN -1 THEN 'Applock request timed out.'
                                    WHEN -2 THEN 'Applock request canceled.'
                                    WHEN -3 THEN 'Applock involved in deadlock'
                                    ELSE 'Parameter validation or other call error.'
                                           END;
                        THROW 50000,@errorMessage,16;
                    END;
            END;

    --get the next primary key value. Reality case is a far more complex number generator
    --that couldn't be done with a sequence or identity
    DECLARE @applockId int ;
    SET @applockId = COALESCE((SELECT MAX(applockId) FROM applock),0) + 1 ;

    --delay for parameterized amount of time to slow down operations
    --and guarantee concurrency problems
    WAITFOR DELAY @stepDelay;

    --insert the next value
    INSERT INTO applock(applockId, connectionId)
    VALUES (@applockId, @connectionId);

    --won't have much effect on this code, since the row will now be
    --exclusively locked, and the max will need to see the new row to
    --be of any effect.
```


www.it-ebooks.info

```
    IF @useApplockFlag = 1 --turns on and off the applock for testing
        EXEC @retval = sp_releaseapplock @Resource = 'applock$test';

    --this releases the applock too
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    --if there is an error, roll back and display it.
    IF @@trancount > 0
        ROLLBACK transaction;
        SELECT CAST(ERROR_NUMBER() AS varchar(10)) + ':' + ERROR_MESSAGE();
END CATCH
```

Now, you can set up a few connections using this stored procedure, attempting multiple connections first without the application lock and then with it. Since we're running the procedure in such a tight loop, it is not surprising that two connections will often get the same value and try to insert new rows using that value when not using the APPLOCK:

```
--test on multiple connections
WAITFOR TIME '21:47'; --set for a time to run so multiple batches
                      --can simultaneously execute
go
EXEC applock$test @connectionId = @@spid
              ,@useApplockFlag = 0 -- <1=use applock, 0 = don't use applock>
              ,@stepDelay = '00:00:00.001'--'delay in hours:minutes:seconds.parts of seconds';
GO 10000 --runs the batch 10000 times in SSMS
```

You will probably be amazed at how many clashes you get when you have application locks turned off. Doing 10,000 iterations of this procedure on three connections on a Core2 Duo, 1.3 GHz-laptop with 0 for the APPLOCK parameter, I got over 1,000 clashes pretty much constantly (evidenced by an error message: `2627:Violation of PRIMARY KEY constraint. . . Cannot insert duplicate key in object 'dbo.applock'. . .`). With application locks turned on, all rows were inserted in a slightly longer than the original time, without any clashes.

To solidify the point that every connection has to follow the rules, turn off application locks on a connection or two and see the havoc that will result. The critical section will now no longer be honored, and you will get tons of clashes quickly, especially if you use the `@stepDelay` parameter to slow the process down.

This is not the only method of implementing the solution to the incrementing values problem. The more common method is to change the code where you get the maximum value to increment and apply locking hints:

```
SET @applockId =
      COALESCE((SELECT MAX(applockId) FROM APPLOCK WITH (UPDLOCK,PAGLOCK)),0) + 1;
```

Changing the code to do this will cause update locks to be held because of the UPDLOCK hint, and the PAGLOCK hint causes page locks to be held (SQL Server can ignore locks when a row is locked and it has not been modified, even if the row is exclusively locked).

The solution I presented is a very generic one for single threading a code segment in T-SQL code, allowing that the one procedure is the only one single threading. It does not take any locks that will block others until it needs to update the data (if there is no changing of data, it won't block any other users, *ever*). This works great for a hotspot where you can clearly cordon off the things being utilized at a given level, like in this example, where all users of this procedure are getting the maximum of the same rows.

www.it-ebooks.info

Optimistic Locking

The opposite of pessimistic locking is optimistic locking (I make this statement merely to win the obvious statement of the year award). Here, the premise is simply to assume that the likelihood of users stepping on one another is limited. Instead of locking resources, locks are only taken during actual data-modification activities, because most of the time, users just look around, and even if the data they're looking at is slightly out of date, it won't hurt anything (this is where the `SNAPSHOT` isolation level is perfect). In the case where two users do try to update the same data, we usually do something to make sure that one user's updates don't overwrite the others. Generally speaking, when the likelihood of two users editing the same data is reasonably low, an optimistic locking scheme can be extremely good for increasing concurrent access to the data. If this is not the case, some form of pessimistic locking might be a better choice.

As an example, it is very unlikely that the same person is calling into your sales call center on two lines talking to two different users. Even scenarios like giving a customer an inventory amount is a place where a sort of optimistic lock is acceptable. If you tell a customer that you have 1,000 of some item on hand and Joey Bigbucks walks up and buys all 1,000 of them, your first customer will be left out in the cold. Use a pessimistic lock (like the application lock example) when you need to implement that critical section of the code in which inventory is being decremented (or even selectively implement a pessimistic lock when your inventory levels are getting low, so you can tell the other customer that you *might* have inventory).

The idea behind optimistic locking is that, in all cases, the data is only locked at the point where the user modifies the data. Data is protected in the server using constraints, triggers, and so on. Choose the best isolation level depending on how important perfection is. That's because, as noted in the "Isolation Levels" section, the default of `READ UNCOMMITTED` is flawed because for some amount of time (hopefully milliseconds), it leaves open the possibility that one user can change data on which your transaction is dependent. For the most part, using the default is considered appropriate, because it greatly enhances concurrency, and the probability of someone modifying data that your transaction is reliant on is close to the chances of being hit by lightning on ten sunny days in a row. It could happen, but it's a slim chance.

Thinking back to the normal progression of events when a user works in an application: the user fetches data, modifies data, and finally, commits data to the database. There can easily be a long interval between fetching the data and committing the changes to the database. In fact, it's also possible that other users could have also fetched and modified the data during the same period of time. Because of this, you need to implement some method to make sure that the data that the client originally fetched matches the data that's stored in the database. Otherwise, the new changes could trample important changes made by another user.

Instead of locking the data by using SQL Server locks, simply employ one of the following schemes to leave data unlocked after it has been fetched to the client and the user makes desired changes:

- *Unchecked:* Just let it happen. If two users modify the same row in the database, the last user wins. This is not the best idea, as the first user might have had something important to say, and this method rejects the first user's changes. I won't cover this any further because it's straightforward. Note that many systems use no locking mechanism at all and just let clashes happen, and their users are as happy as they can be. Such an approach is never what I suggest for controlling your important data resources, but so far, I have not been elected data emperor, just data architect.
- *Row-based:* Protect your data at the row level by checking to see if the rows being modified are the same as in the table. If not, refresh the data from the table, showing the user what was changed. When optimistic locking is actually implemented, this is by far the most common method used.

www.it-ebooks.info

- *Logical unit of work:* A logical unit of work is used to group a parent record with all its child data to allow a single optimistic lock to cover multiple tables. For example, you'd group an invoice and the line items for that invoice. Treat modifications to the line items the same way as a modification to the invoice, for locking purposes.

Although it isn't typically a good idea to ignore the problem of users overwriting one another altogether, this is a commonly *decided* upon method for some companies. On the other hand, the best plan is optimally a mixture of the row-based solution for most tables and a logical unit of work for major groups of tables that make up some common object.

In the following sections, I'll cover row-based locking and the logical unit of work. The unchecked method ignores the concern that two people might modify the same row twice, so there's no coding (or thinking!) required.

Row-Based Locking

You must implement a row-based scheme to check on a row-by-row basis whether or not the data that the user has retrieved is still the same as the one that's in the database. The order of events now is fetch data, modify data, check to see that the row (or rows) of data are still the same as they were, and then commit the changes.

There are three common methods to implement row-based optimistic locking:

- *Check all columns in the table:* If you cannot modify the table structure, which the next two methods require, you can check to make sure that all the data you had fetched is still the same and then modify the data. This method is the most difficult, because any modification procedure you write must contain parameters for the previous values of the data, which isn't a good idea. Checking all columns is useful when building bound data-grid types of applications, where there are direct updates to tables, especially if not all tables can follow the rather strict rules of the next two methods.
- *Add a time column to the table:* Set the `point in time` value when the row is inserted and subsequently updated. Every update to the table is required to modify the value in the table to set the `rowLastModifiedTime` column. Generally, it's best to use a trigger for keeping the column up to date, and often, it's nice to include a column to tell which user last modified the data (you need someone to blame!). Later in this section, I'll demonstrate a simple `INSTEAD OF` trigger to support this approach.
- *Use a* `rowversion` *(also known as* `timestamp`*) column:* In the previous method, you used a manually controlled value to manage the optimistic lock value. This method uses column with a `rowversion` datatype. The `rowversion` datatype automatically gets a new value for every command used to modify a given row in a table.

The next two sections cover adding the optimistic lock columns to your tables and then using them in your code.

Adding Optimistic Lock Columns

In this section, we'll add an optimistic lock column to a table to support either adding the `datetime` column or the `rowversion` column. The first method mentioned, checking all columns, needs no table modifications.

As an example, let's create a new simple table, in this case `hr.person` (again, use any database you like; the sample uses `tempdb`). Here's the structure:

www.it-ebooks.info

```
CREATE SCHEMA hr;
GO
CREATE TABLE hr.person
(
     personId int NOT NULL IDENTITY(1,1) CONSTRAINT PKperson primary key,
     firstName varchar(60) NOT NULL,
     middleName varchar(60) NOT NULL,
     lastName varchar(60) NOT NULL,

     dateOfBirth date NOT NULL,
     rowLastModifyTime datetime2(3) NOT NULL
         CONSTRAINT DFLTperson_rowLastModifyTime DEFAULT (SYSDATETIME()),
     rowModifiedByUserIdentifier nvarchar(128) NOT NULL
         CONSTRAINT DFLTperson_rowModifiedByUserIdentifier default SUSER_SNAME()
);
```

Note the two columns for our optimistic lock, named `rowLastModifyTime` and `rowModifiedByUserIdentifier`. I'll use these to hold the last date and time of modification and the SQL Server's login name of the principal that changed the row. There are a couple ways to implement this:

- *Let the manipulation layer manage the value like any other column:* This is often what client programmers like to do, and it's acceptable, as long as you're using trusted computers to manage the timestamps. I feel it's inadvisable to allow workstations to set such values, because it can cause confusing results. For example, say your application displays a message stating that another user has made changes, and the time the changes were made is in the future, based on the client's computer. Then the user checks out his or her PC clock, and it's set perfectly.
- *Using SQL Server code:* For the most part, triggers are implemented to fire on any modification to data.

As an example of using SQL Server code (my general method of doing this), I'll implement an `INSTEAD OF` trigger on the update of the `hr.person` table (note that the `errorLog$insert` procedure was created back in Chapter 6 and has been commented out for this demonstration in case you don't have it available):

```
CREATE TRIGGER hr.person$InsteadOfUpdate
ON hr.person
INSTEAD OF UPDATE AS
BEGIN

    --stores the number of rows affected
    DECLARE @rowsAffected int = @@rowcount,
            @msg varchar(2000) = ''; --used to hold the error message

    SET NOCOUNT ON; --to avoid the rowcount messages
    SET ROWCOUNT 0; --in case the client has modified the rowcount

    BEGIN TRY
          --[validation blocks]
          --[modification blocks]
          --remember to update ALL columns when building instead of triggers
          UPDATE hr.person
          SET    firstName = inserted.firstName,
```


www.it-ebooks.info

```
                middleName = inserted.middleName,
                lastName = inserted.lastName,
                dateOfBirth = inserted.dateOfBirth,

                rowLastModifyTime = DEFAULT, -- set the value to the DEFAULT
                rowModifiedByUserIdentifier = DEFAULT

                FROM hr.person
                    JOIN inserted
                            on hr.person.personId = inserted.personId;
    END TRY
      BEGIN CATCH
              IF @@trancount > 0
                  ROLLBACK TRANSACTION;

              THROW; --will halt the batch or be caught by the caller's catch block

      END CATCH
END
```

Then, insert a row into the table:

```
INSERT INTO hr.person (firstName, middleName, lastName, dateOfBirth)
VALUES ('Paige','O','Anxtent','19391212');

SELECT *
FROM   hr.person;
```

Now, you can see that the data has been created:

```
personId        firstName       middleName      lastName        dateOfBirth
--------        --------        ----------      --------        -----------
1               Paige           O               Anxtent         1939-12-12

rowLastModifyTime               rowModifiedByUserIdentifier
-----------------------         -------------------------
2012-02-19 22:29:56.389         DENALI-PC\AlienDrsql
```

Next, update the row:

```
UPDATE  hr.person
SET     middleName = 'Ona'
WHERE   personId = 1;

SELECT  rowLastModifyTime
FROM    hr.person;
```

You should see that the update date has changed (in my case, it was pretty doggone late at night, but such is the life):

```
rowLastModifyTime
-----------------------
2012-02-19 22:30:41.664
```


www.it-ebooks.info

If you want to set the value on insert, or implement `rowCreatedByDate` or `userIdentifier` columns, the code would be similar. Because this has been implemented in an `INSTEAD OF` trigger, the user or even the programmer cannot overwrite the values, even if they include it in the column list of an `INSERT`.

As previously mentioned, the other method that requires table modification is to use a `rowversion` column. In my opinion, this is the best way to go, and I almost always use a `rowversion`. I usually have the row modification columns on there as well, for the user's benefit. I find that the modification columns take on other uses and have a tendency to migrate to the control of the application developer, and `rowversion` columns never do. Plus, even if the triggers don't make it on the table for one reason or another, the `rowversion` column continues to work. Sometimes, you may be prohibited from using `INSTEAD OF` insert triggers for some reason (recently, I couldn't use them in a project I worked on because they invalidate the `identity` functions).

■ **Tip** You might know this as the `timestamp` datatype, which is what it has been named since the early days of SQL Server. In the ANSI standard, a `timestamp` column is a date and time value. `rowversion` is a much better name for the datatype.

Let's add a `rowversion` column to our table to demonstrate using it as an optimistic lock:

```
ALTER TABLE hr.person
    ADD rowversion rowversion;
GO
SELECT personId, rowversion
FROM   hr.person;
```

You can see now that the `rowversion` has been added and magically updated:

```
personId        rowversion
--------        ------------------
1               0x00000000000007D6
```

Now, when the row gets updated, the `rowversion` is modified:

```
UPDATE hr.person
SET    firstName = 'Paige' --no actual change occurs
WHERE  personId = 1;
```

Then, looking at the output, you can see that the value of the `rowversion` has changed:

```
SELECT personId, rowversion
@@FROM hr.person;
```

This returns the following result:

```
personId        rowversion
--------        ------------------
1               0x00000000000007D7
```


www.it-ebooks.info

■ **Caution** You aren't guaranteed anything when it comes to `rowversion` values, other than that they'll be unique in a database. Avoid using the value for any reason other than to tell when a row has changed.

Coding for Row-Level Optimistic Locking

Next, include the checking code in your stored procedure. Using the `hr.person` table previously created, the following code snippets will demonstrate each of the methods (note that I'll only use the optimistic locking columns germane to each example and won't include the others).

Check all the cached values for the columns:

```
UPDATE hr.person
SET    firstName = 'Headley'
WHERE  personId = 1 --include the key
  and  firstName = 'Paige'
  and  middleName = 'ona'
  and  lastName = 'Anxtent'
  and  dateOfBirth = '19391212';
```

It's a good practice to check your rowcount after an update with an optimistic lock to see how many rows have changed. If it is 0, you could check to see if the row exists with that primary key:

```
IF EXISTS ( SELECT *
            FROM   hr.person
            WHERE  personId = 1) --check for existence of the primary key
  --raise an error stating that the row no longer exists
ELSE
   --raise an error stating that another user has changed the row
```

Use a `date` column:

```
UPDATE hr.person
SET    firstName = 'Fred'
WHERE  personId = 1 --include the key
  AND  rowLastModifyTime = ' 2012-02-19 22:33:53.845';
```

Use a `rowversion` column:

```
UPDATE hr.person
SET    firstName = 'Fred'
WHERE  personId = 1
  AND  rowversion = 0x00000000000007D9;
```

Which is better performance-wise? Either of these generally performs just as well as the other, because in all cases, you're going to be using the primary key to do the bulk of the work fetching the row and then your update. There's a bit less overhead with the last two columns, because you don't have to pass as much data into the statement, but that difference is negligible.

Deletions use the same `WHERE` clause, because if another user has modified the row, it's probably a good idea to see if that user's changes make the row still valuable:

```
DELETE FROM hr.person
```


www.it-ebooks.info

```
WHERE       personId = 1
  And       rowversion = 0x00000000000007D9;
```

However, if the timestamp had changed since the last time the row was fetched, this would delete zero rows, since if someone has modified the rows since you last touched them, perhaps the deleted row now has value. I typically prefer using a `rowversion` column because it requires the least amount of work to always work perfectly. On the other hand, many client programmers prefer to have the manipulation layer of the application set a `datetime` value, largely because the `datetime` value has meaning to them to let them see when the row was last updated. Truthfully, I, too, like keeping these automatically modifying values in the table for diagnostic purposes. However, I prefer to rely on the `rowversion` column for locking because it is far simpler and safer and cannot be overridden by any code, no matter how you implement the other columns.

Logical Unit of Work

Although row-based optimistic locks are helpful, they do have a slight downfall. In many cases, several tables together make one "object." A good example is an invoice with line items. The idea behind a logical unit of work is that, instead of having a row-based lock on the invoice and all the line items, you might only implement one on the invoice and use the same value for the line items. This strategy does require that the user always fetch not only the invoice line items but at least the invoice's timestamp into the client's cache when dealing with the invoice line items. Assuming you're using a `rowversion` column, I'd just use the same kind of logic as previously used on the `hr.person` table. In this example, we'll build the procedure to do the modifications.

When the user wants to insert, update, or delete line items for the invoice, the procedure requires the `@objectVersion` parameter and checks the value against the invoice, prior to update. Consider that there are two tables, minimally defined as follows:

```
CREATE SCHEMA invoicing;
GO
--leaving off who invoice is for, like an account or person name
CREATE TABLE invoicing.invoice
(
     invoiceId int NOT NULL IDENTITY(1,1),
     number varchar(20) NOT NULL,
     objectVersion rowversion NOT NULL,
     CONSTRAINT PKinvoicing_invoice PRIMARY KEY (invoiceId)
);

--also forgetting what product that the line item is for
CREATE TABLE invoicing.invoiceLineItem

(
     invoiceLineItemId int NOT NULL,
     invoiceId int NULL,
     itemCount int NOT NULL,
     cost int NOT NULL,
      CONSTRAINT PKinvoicing_invoiceLineItem PRIMARY KEY (invoiceLineItemId),
      CONSTRAINT FKinvoicing_invoiceLineItem$references$invoicing_invoice
           FOREIGN KEY (invoiceId) REFERENCES invoicing.invoice(invoiceId)
);
```

For our delete procedure for the invoice line item, the parameters would have the key of the invoice and the line item, plus the `rowversion` value:

www.it-ebooks.info

```
CREATE PROCEDURE invoiceLineItem$del
(
    @invoiceId int, --we pass this because the client should have it
                    --with the invoiceLineItem row
    @invoiceLineItemId int,
    @objectVersion rowversion
) as
  BEGIN
    --gives us a unique savepoint name, trim it to 125
    --characters if the user named it really large
    DECLARE @savepoint nvarchar(128) =
                          CAST(OBJECT_NAME(@@procid) AS nvarchar(125)) +
                                  CAST(@@nestlevel AS nvarchar(3));
    --get initial entry level, so we can do a rollback on a doomed transaction
    DECLARE @entryTrancount int = @@TRANCOUNT;

    BEGIN TRY
        BEGIN TRANSACTION;
        SAVE TRANSACTION @savepoint;

        UPDATE invoice
        SET    number = number
        WHERE  invoiceId = @invoiceId
          And  objectVersion = @objectVersion;

        DELETE invoiceLineItem
        FROM   invoiceLineItem
        WHERE  invoiceLineItemId = @invoiceLineItemId;

        COMMIT TRANSACTION;

    END TRY
    BEGIN CATCH

        --if the tran is doomed, and the entryTrancount was 0,
        --we have to roll back
        IF XACT_STATE()= -1 and @entryTrancount = 0
            ROLLBACK TRANSACTION;
        --otherwise, we can still save the other activities in the
        --transaction.
        ELSE IF xact_state() = 1 --transaction not doomed, but open
           BEGIN
               ROLLBACK TRANSACTION @savepoint;
               COMMIT TRANSACTION;
           END

    DECLARE @ERRORmessage nvarchar(4000)
    SET @ERRORmessage = 'Error occurred in procedure ''' +
          OBJECT_NAME(@@PROCID) + ''', Original Message: '''
          + ERROR_MESSAGE() + '''';
```


www.it-ebooks.info

```
    THROW 50000,@ERRORmessage,16;
    RETURN -100;

    END CATCH
END
```

Instead of checking the `rowversion` on an `invoiceLineItem` row, we check the `rowversion` (in the `objectVersion` column) on the `invoice` table. Additionally, we must update the `rowversion` value on the `invoice` table when we make our change, so we update the `invoice` row, simply setting a single column to the same value. There's a bit more overhead when working this way, but it's normal to update multiple rows at a time from the client. You'd probably want to architect your solution with multiple procedures, one to update and check the optimistic lock and others to do the insert, update, and delete operations.

■ **Tip** Using table parameters, you could build a single procedure that accepted a list of ID values as parameters that included `rowversion` values quite easily. This would be yet another way to implement proper optimistic locking on a group of rows.

Best Practices

The number-one issue when it comes to concurrency is data quality. Maintaining consistent data is why you go through the work of building a database in the first place. Generally speaking, if the only way to get consistent results was to have every call single threaded, it would be worth it. Of course, we don't have to do that except in rare situations, and SQL Server gives us tools to make it happen with the isolation levels. Use them as needed. It's the data that matters:

- *Use transactions as liberally as needed:* It's important to protect your data, 100% of the time. Each time data is modified, enclosing the operation in a transaction isn't a bad practice. This gives you a chance to check status, number of rows modified, and so on, and if necessary, to roll back the modification.
- *Keep transactions as short as possible:* The smaller the transaction, the less chance there is of it holding locks. Try not to declare variables, create temporary tables, and so on inside a transaction unless doing so necessary. Make sure that all table access within transactions is required to be executed as an atomic operation.
- *Recognize the difference between hardware limitations and SQL Server concurrency issues:* If the hardware is maxed out (excessive disk queuing, 90%–plus CPU usage, and so on), consider adding more hardware. However, if you're single-threading calls through your database because of locking issues, you could add 20 processors and a terabyte of RAM and still see only marginal improvement.
- *Fetch all rows from a query as fast as possible:* Depending on the isolation level and editability of the rows being returned, locks held can interfere with other users' ability to modify or even read rows.
- *Make sure that all queries use reasonable execution plans:* The better all queries execute, the faster the queries will execute, and it follows that locks will be held for a shorter amount of time. Remember, too, that scans will require every row in the table to be locked, whereas a seek will lock far fewer rows.

www.it-ebooks.info

- *Use some form of optimistic locking mechanism:* Use optimistic locking, preferably using a `rowversion` column, because it requires the smallest amount of coding and is managed entirely by SQL Server. The only code that's required when programming is to validate the value in the `rowversion` column.
- *Consider using some form of the SNAPSHOT isolation level:* Either code all your optimistic-locked retrieval operations with `SET SNAPSHOT ISOLATION LEVEL`, or change the database setting for `READ_COMMITTED_SNAPSHOT` to `ON`. This alters how the `READ COMMITTED` isolation level reads snapshot information at the statement level. Be careful to test existing applications if you're going to make this change, because these settings do alter how SQL Server works and might negatively affect how your programs work. I suggest using full `SNAPSHOT` isolation level for read-only operations anyhow, if it's reasonable for you to do so.

Summary

Concurrency is an important topic, and a difficult one. It seems easy enough: keep the amount of time a user needs to be in the database to a minimum; have adequate resources on your machine.

The fact is that concurrency is a juggling act for SQL Server, Windows, the disk system, the CPUs, and so on. If you have reasonable hardware for your situation, use the `SNAPSHOT` isolation level for retrieval and `READ COMMITTED` for other calls, and you should have no trouble with large-scale blocking on your server. This solution sounds perfect, but the greater the number of users, the more difficult a time you'll have making things perform the way you want. Concurrency is one of the fun jobs for a DBA, because it's truly a science that has a good deal of artsy qualities. You can predict only so much about how your user will use the system, and then experience comes in to tune queries, tune hardware, and tweak settings until you have them right.

I discussed some of the basics of how SQL Server implements controls to support concurrent programming, such that many users can be supported using the same data with locks and transactions. Then, I covered isolation levels, which allow you to tweak the kinds of locks taken and how long they're held on a resource. The most important part of this chapter was the part on optimistic locking. Because the trend for implementing systems is to use cached data sets, modify that set, and then flush it back, you must make sure that other users haven't made changes to the data while it's cached.

www.it-ebooks.info

CHAPTER 12

Reusable Standard Database Components

One little spark, of inspiration, is at the heart, of all creation. Right at the start, of everything that's new. One little spark, lights up for you.

— The Sherman Brothers for Disney

As we near the end of the database design process, the database is pretty much completed from a design standpoint. We have spent time looking at performance, concurrency and security patterns that you can follow to help implement the database in a manner that will work well under most any typical OLTP style load. In this chapter (and again somewhat in the next two), we are going to look at applying "finishing touches" to the database that can be used to enhance the user experience and assist with the querying and maintaining of the database. We won't flesh out all of the details for every concept in this chapter, because a lot of the examples include parts of the system that are decidedly more aligned to the DBA than the architect/programmer (or are just far too large to fit in this book). However, the point I will be making make here is that our goal will be end up with a self contained database container with as much of the database coding and maintenance functionality as possible. In Chapter 9, we introduced the concept of a contained database to help maintain a secure and portable database, and here I will present some additional add-in capabilities and expansion point ideas to add to make your database that much more usable.

The reality of database design is that most databases are rarely cookie-cutter affairs. Most companies, even when they buy a third-party package to implement some part of their business, are going to end up making (in many cases substantial) customizations to the database to fit their needs. If you are starting a very large project, you may even want to look at previous models or perhaps pre-built "universal data models," such as those in Len Silverston's series of books, the first of which is *The Data Model Resource Book: A Library of Universal Data Models for All Enterprises* (Wiley, 2001) (perhaps the only book on database design with a larger title that the book you hold in your hands right now.) Karen Lopez (`@datachick` on Twitter) frequently speaks on the subject of universal models in the SQL PASS universe of presenters that I am generally involved with. Even these universal models may only be useful as starting points to help you map from your "reality" to a common view of a given sort of business.

In this chapter, however, I want to explore the parts of the database that I find to be useful and almost always the same for every database I create. Not every database will contain all of what I will present, but when I need a common feature I will use the exact same code in every database, with the obvious caveat that I am constantly looking for new ways to improve almost everything I use over time (not to mention it gives me something to write

www.it-ebooks.info

about when I run out of Lego sets to build). Hence, sometimes a database may use an older version of a feature until it can be upgraded. I will cover the following topics:

- *Numbers table*: A table of numbers, usually integers that can be used for a number of interesting uses (not many of the mathematics-related).
- *Calendar table*: A table where every row represents a day, assisting in grouping data for queries, both for reports and operational usage.
- *Utility objects*: Every programmer has code that they use to make their job easier. Utilities to monitor usage of the system; extended DDL to support operations that aren't part of the base DDL in T-SQL.
- *Logging objects*: Utilities to log the actions of users in the database, generally for system management reasons. A common use is an error log to capture when and where errors are occurring.
- *Other possibilities*: In this section, I will present a list of additional ideas for ways to extend your databases in ways that will give you independent databases that have common implementation.

Not every database can look alike, even two that do almost the exact same thing will rarely be all that alike unless the designer is the same, but following the patterns of implementation we have discussed all throughout the book thus far and the practices we will discuss in this chapter, we can produce databases that are similar enough such that the people supporting your work will have it easy figuring out what you had in mind.

If you are dealing with a third party system where it is forbidden to add any of your own objects, even in a schema that is separated from the shipped schemas, don't think that everything I am saying here doesn't apply to you. All of the example code presented supposes a single database approach. In such cases a common approach is to create a companion database where you locate code you need to access their code from the database tier. Some examples would need to be slightly reworked for that model, but that rework would be minimal.

■ **Note** For the examples in this chapter, I am going to use a copy of the `Adventureworks2012` database to stick to the supposition of the chapter that we should place the tables in the database with the data you are working with. If you are working with a community version of `AdventureWorks2012` that you cannot modify, you can build your own companion database for the examples. I will include a comment in the query to note where the data is specifically from that database. In cases where cross database access will not be trivial, I will note that in the code with a comment.

Numbers Table

A numbers table is a precalculated table of numbers, primarily non-negative integers, which you can use for some purpose. The name "numbers" is pretty open ended, but getting so specific as `nonNegativeIntegers` is going to get you ridiculed by the other programmers on the playground. In previous editions of the book I have used the name sequence, but with the addition of the sequence object, the name "numbers" was the next best thing. We will use the numbers table when we need to work with data in an ordered manner, particularly a given sequence of numbers. For example, if you needed a list of the top ten products sold and you only sold six, you would have to somehow manufacture four more rows for display. Having a table where you can easily output a sequence of numbers is going to be a very valuable asset at times indeed.

While you can make the numbers table contain any type of numbers you may need, usually it is just a simple table of non-negative integers from 0 to some reasonable limit, where reasonable is more or less how many

www.it-ebooks.info

you find you need. I generally load mine by default up to 99999 (99999 gives you full five digits (and is a very convenient number for the query I will use to load the table.) With the algorithm I will present, you can easily expand to create a sequence of numbers that is larger than you can store in SQL Server.

There are two really beautiful things behind this concept. First, the table of non-negative integers has some great uses dealing with text data, as well as doing all sorts of math with. Second, you can create additional attributes or even other sequence tables that you can use to represent other sets of numbers that you find useful or interesting. For example:

- Even or odd, prime, squares, cubes, and so on
- Other ranges or even other grains of values, for example, (−1, −.5, 0, .5, 1)
- Letters of the alphabet

In the examples in this section, we will look at several techniques you may find useful, and possibly quite often. The code to generate a simple numbers table of integers is pretty simple; though it looks a bit daunting the first time you see it. It is quite fast to execute in this form, but no matter how fast it may seem, it is not going to be faster than querying from a table that has the sequence of numbers precalculated and stored ahead of time.

```
;WITH digits (I) AS
            (--set up a set of numbers from 0-9
             SELECT I
             FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS digits (I))
,integers (I) AS (
        SELECT D1.I + (10*D2.I) + (100*D3.I) + (1000*D4.I)
              -- + (10000*D5.I) + (100000*D6.I)
        FROM digits AS D1 CROSS JOIN digits AS D2 CROSS JOIN digits AS D3 CROSS JOIN digits AS D4
              --CROSS JOIN digits AS D5 CROSS JOIN digits AS D6
        )
SELECT I
FROM integers
ORDER BY I;
```

This code will return a set of 10,000 rows, as follows:

```
I
-----
0
1
2
…
9998
9999
```

Uncommenting the code for the D5 and D6 tables will give you an order of magnitude increase for each, up to 999,999 rows. The code itself is pretty interesting (this isn't a coding book, but it is a really useful technique). Breaking the code down, you get the following:

```
;WITH digits (I) AS
        (--set up a set of numbers from 0-9
            SELECT I
            (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS digits (I))
```


www.it-ebooks.info

This is just simply a set of ten rows from 0 to 9. The next bit is where the true brilliance begins. (No, I am not claiming I came up with this. I first saw it on Erland Sommarskog's web site a long time ago, using a technique I will show you in a few pages to split a comma-delimited string.) You cross-join the first set over and over, multiplying each level by a greater power of 10. The result is that you get one permutation for each number. For example, since 0 is in each set, you get one permutation that results in 0. You can see this better in the following smallish set:

```
;WITH digits (I) AS (--set up a set of numbers from 0-9
        SELECT i
        FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS digits (I))
SELECT D1.I AS D1I, 10*D2.I AS D2I, D1.I + (10*D2.I) AS [Sum]
FROM digits AS D1 CROSS JOIN digits AS D2
ORDER BY [Sum];
```

This returns the following, and you can see that by multiplying the D2.I value by 10, you get the ten's place repeated, giving you a very powerful mechanism for building a large set. In the full query, each of the additional digit table references have another power of ten in the SELECT clause multiplier, allowing you to create a very large set (rows removed and replaced with . . . for clarity and to save a tree):

```
D1I         D2I         Sum
----        ----        -----
0           0           0
1           0           1
2           0           2
3           0           3
4           0           4
5           0           5
6           0           6
7           0           7
8           0           8
9           0           9
0           10          10
1           10          11
2           10          12
3           10          13
4           10          14
…
6           80          86
7           80          87
8           80          88
9           80          89
0           90          90
1           90          91
2           90          92
3           90          93
4           90          94
5           90          95
6           90          96
7           90          97
8           90          98
9           90          99
```


www.it-ebooks.info

This kind of combination of sets is a very useful technique in relational coding. As I said earlier, this isn't a query book, but I feel it necessary to show you the basics of why this code works, because it is a very good mental exercise. Using the full query, you can create a sequence of numbers that you can use in a query.

So, initially create a simple table named Number with a single column I (because it is a typical value used in math to denote an index in a sequence, such as x_i, where the I denotes a sequence of values of x. The primary purpose of the Numbers is to introduce an ordering to a set to assist in an operation.). I will create this table in a schema named Tools to contain the types of tool objects, functions, and procedures we will build in this chapter. In all likelihood, this is a schema you would grant EXECUTE and SELECT to public and make the tools available to any user you have given ad hoc query access to.

```
USE AdventureWorks2012;
GO
CREATE SCHEMA Tools;
GO
CREATE TABLE Tools.Number
(
      I int NOT NULL CONSTRAINT PKTools_Number PRIMARY KEY
);
```

Then I will load it with integers from 0 to 99999.

```
;WITH digits (I) AS (--set up a set of numbers from 0-9
        SELECT I
        FROM (VALUES (0),(1),(2),(3),(4),(5),(6),(7),(8),(9)) AS digits (I))
--builds a table from 0 to 99999
,Integers (I) AS (
        SELECT D1.I + (10*D2.I) + (100*D3.I) + (1000*D4.I) + (10000*D5.I)
          --+ (100000*D6.I)
        FROM digits AS D1 CROSS JOIN digits AS D2 CROSS JOIN digits AS D3
          CROSS JOIN digits AS D4 CROSS JOIN digits AS D5
          /* CROSS JOIN digits AS D6 */)
INSERT INTO Tools.Number(I)
SELECT  I
FROM    Integers;
```

So if you wanted to count the integers between 1 and 1000 (inclusive), it is as simple as:

```
SELECT COUNT(*)
FROM   Tools.Number
WHERE  I BETWEEN 1 AND 1000;
```

Of course, that would be a bit simplistic, and there had better be 1000 values between (inclusive) 1 and 1000, but what if you wanted the number of integers between 1 and 1000 that are divisible by 9 or 7?

```
SELECT COUNT(*)
FROM   Tools.Number
WHERE  I BETWEEN 1 AND 1000
    AND (I % 9 = 0 OR I % 7 = 0);
```

This returns the *obvious* answer: 238. Sure, a math nerd could sit down and write a formula to do this, but why? And if you find you need these values quite often, you could create a table called Tools.DivisibleByNineAndSevenNumber, or add columns to the number table called DivisibleByNineFlag and DivisibleBySevenFlag if it were needed in context of integers that were not divisible by 9 or 7. The simple numbers table is the most typical need, but you can make a table of any sorts of numbers that you need (prime numbers? squares? cubes?). The last example of this chapter is an esoteric example of what you can do with a

www.it-ebooks.info

table of numbers to do some pretty (nerdy) fun stuff with a table of numbers, but for OLTP use, the goal will be (as we discussed in Chapter 5 on normalization) to pre-calculate values *only* when they are used often and can never change. Numbers-type tables are an excellent candidate for storing pre-calculated values because the set of integer numbers and prime numbers are the same now as back in 300 BC when Euclid was working with them.

In this section, I will present the following uses of the numbers table to get you going:

- *Determining the contents of a string*: Looking through a string without looping by using the ordered nature of the numbers table to manage the iteration using relational code.
- *Determining gaps in a sequence*: Having a set of data that contains all of the values allows you to use relational subtraction to find missing values.
- *Separating comma-delimited items*: Sometimes data is not broken down into scalar values like you desire.
- *Stupid mathematic tricks*: I take the numbers table to abstract levels, solving a fairly complex math problem that, while not terribly applicable in a practical manner, serves as an experiment to build upon if you have similar, complex problem-solving needs.

Determining the Contents of a String

As a fairly common example usage, it sometimes occurs that a value in a string you are dealing with is giving your code fits, but it isn't easy to find what the issue is. If you want to look at the Unicode (or ASCII) value for every character in a string, you can do something like the following:

```
DECLARE @string varchar(20) = 'Hello nurse!';

SELECT  Number.I AS Position,
        SUBSTRING(split.value,Number.I,1) AS [Character],
        UNICODE(SUBSTRING(split.value,Number.I,1)) AS [Unicode]
FROM    Tools.Number
        CROSS JOIN (SELECT @string AS value) AS split
WHERE   Number.I > 0 --No zeroth position
        AND Number.I <= LEN(@string)
ORDER BY Position;
```

This returns the following:

```
Position        Character           Unicode
--------        ---------           --------
1               H                   72
2               e                   101
3               l                   108
4               l                   108
5               o                   111
6                                   32
7               n                   110
8               u                   117
9               r                   114
10              s                   115
11              e                   101
12              !                   33
```


www.it-ebooks.info

This in and of itself is interesting, and sometimes when you execute this, you might see a little square character that can't be displayed and a really large/odd Unicode value (like 20012, picking one randomly) that you didn't expect in your database of English-only words. What really makes the technique awesome is that not only didn't we have to write a routine to go column by column, we won't have to do go row by row either. Using a simple join, you can easily do this for a large number of rows at once, this time joining to a table in the AdventureWorks2012 database that can provide us with an easy example set.

```
SELECT LastName, Number.I AS position,
       SUBSTRING(Person.LastName,Number.I,1) AS [char],
       UNICODE(SUBSTRING(Person.LastName, Number.I,1)) AS [Unicode]
FROM /*Adventureworks2012.*/ Person.Person
    JOIN Tools.Number
        ON Number.I <= LEN(Person.LastName )
            AND UNICODE(SUBSTRING(Person.LastName, Number.I,1)) IS NOT NULL
ORDER BY LastName;
```

This returns 111,969 rows (one for each character in a last name) in only around 3 seconds on a virtual machine hosted on my writing laptop (which is a quite decent Alienware MX11 Core 2 Duo 1.3 GHz; 8 GB; 250 GB, 7200RPM SATA drive; 11.6-inch netbook-sized laptop).

```
LastName        position        char        Unicode
---------       ---------       ------      ---------
Abbas           1               A           65
Abbas           2               b           98
Abbas           3               b           98
Abbas           4               a           97
Abbas           5               s           115
Abel            1               A           65
Abel            2               b           98
Abel            3               e           101
Abel            4               l           108
……              …               .           …
```

With that set, you could easily start eliminating known safe Unicode values with a simple where clause and find your evil outlier that is causing some issue with some process. For example, you could find all names that include a character not in the normal A–Z, space, comma, or dash characters.

```
SELECT LastName, Number.I AS Position,
      SUBSTRING(Person.LastName,Number.I,1) AS [Char],
      UNICODE(SUBSTRING(Person.LastName, Number.I,1)) AS [Unicode]
FROM /*Adventureworks2012.*/ Person.Person
     JOIN Tools.Number
        ON Number.I <= LEN(Person.LastName )
           AND UNICODE(SUBSTRING(Person.LastName, Number.I,1)) IS NOT NULL
--Note I used both a-z and A-Z in LIKE in case of case sensitive AW database
WHERE SUBSTRING(Person.LastName, Number.I,1) NOT LIKE '[a-zA-Z ~''~-]' ESCAPE '~'
ORDER BY LastName, Position;
```


www.it-ebooks.info

This returns the following:

```
LastName        Position          Char          Unicode
---------       ---------         -------       ---------
Mart¡nez        5                 ¡             161
```

This can be a remarkably powerful tool when trying to figure out what data is hurting your application with some unsupported text particularly when dealing with a stream of data from an outside source.

Finding Gaps in a Sequence of Numbers

Another common issue that we have when using a surrogate is that there can be gaps in their values. Ideally, this should not be an issue, but when troubleshooting errors it is often useful to be able to determine the missing numbers in a range. For example, say you have a table with a domain of values between 1 and 10. How might you determine if a value isn't used? This is fairly simple; you can just do a distinct query on the used values and then check to see what values aren't used, right? Well how about if you had to find missing values in 20,000+ distinct values? This is not quite going to work if a lot of values aren't used. For example, consider the Person table in the `AdventureWorks2012` database. Running the following query, you can see that not every `BusinessEntityID` is used.

```
SELECT MIN(BusinessEntityID) AS MinValue, MAX(BusinessEntityID) AS MaxValue,
       MAX(BusinessEntityID) - MIN(BusinessEntityID) + 1 AS ExpectedNumberOfRows,
       COUNT(*) AS NumberOfRows,
       MAX(BusinessEntityID) - COUNT(*) AS MissingRows
FROM /*Adventureworks2012.*/ Person.Person;
```

This returns the following:

```
MinValue    MaxValue    ExpectedNumberOfRows      NumberOfRows      MissingRows
---------   ---------   ---------------------     -------------     -------------
1           20777       20777                     19972             805
```

So we know that there are 805 rows missing between `BusinessEntityID` values 1 and 20777. To discover these rows, we take a set of values from 1 to 20777 with no gaps, and subtract the rows using the `EXCEPT` relational operator:

```
SELECT    Number.I
FROM      Tools.Number
WHERE     I BETWEEN 1 AND 20777
EXCEPT
SELECT    BusinessEntityID
FROM /*Adventureworks2012.*/ Person.Person;
```

Execute this query and you will find that there are 805 rows returned. Using the subtraction method with the Numbers table is a very powerful method that you can use in lots of situations where you need to find what isn't there rather than what is.

www.it-ebooks.info

Separating Comma Delimited Items

My last example that you can translate to a direct business need comes from Erland Sommarskog's web site (www.sommarskog.se/) on arrays in SQL Server, as well as Aaron Bertrand's old ASPFAQ web site. Using this code, you can take a comma-delimited list to return it as a table of values (which is the most desirable form for data in SQL Server in case you have just started reading this book on this very page and haven't learned about normalization yet).

```
DECLARE @delimitedList varchar(100) = '1,2,3'

SELECT SUBSTRING(',' + @delimitedList + ',',I + 1,
      CHARINDEX(',',',' + @delimitedList + ',',I + 1) - I - 1) AS value
FROM Tools.Number
WHERE I >= 1
  AND I < LEN(',' + @delimitedList + ',') - 1
  AND SUBSTRING(',' + @delimitedList + ',', I, 1) = ','
ORDER BY I;
```

This returns the following:

```
Value
------
1
2
3
```

The way this code works is pretty interesting in and of itself. It works by doing a substring on each row. The key is in the WHERE clause.

```
WHERE I >= 1
    AND I < LEN(',' + @delimitedList + ',') - 1
    AND SUBSTRING(',' + @delimitedList + ',', i, 1) = ','
```

The first line is there because SUBSTRING starts with position 1. The second limits the rows in Tools.Number to more than the length of the @delimitedList variable. The third includes rows only where the SUBSTRING of the value at the position returns the delimiter, in this case, a comma. So, take the following query:

```
DECLARE @delimitedList varchar(100) = '1,2,3';

SELECT I
FROM Tools.Number
WHERE I >= 1
    AND I < LEN(',' + @delimitedList + ',') - 1
    AND SUBSTRING(',' + @delimitedList + ',', I, 1) = ','
ORDER BY I;
```

Executing this, you will see the following results, showing you the position of each value in the list:

```
Value
-----
1
3
5
```


www.it-ebooks.info

Since the list has a comma added to the beginning and end of it in the query, you will see that that the positions of the commas are represented in the list. The SUBSTRING in the SELECT clause of the main query simply fetches all of the @delimitedList value up to the next comma. This sort of use of the sequence table will allow you to do what at first seems like it would require a massive, iterating algorithm in order to touch each position in the string individually (which would be slow in T-SQL, though you might get away with it in the CLR) and does it all at once in a set-based manner that is actually very fast.

Finally, I have expanded on this technique to allow you to do this for every row in a table that needs it by joining the Tools.Numbers table and joining on the values between 1 and the length of the string (and delimiters). The best use for this code is to normalize a set of data where some programmer thought it was a good idea to store data in a comma-delimited list (it rarely is) so that you can use proper relational techniques to work with this data. It is actually pretty sad how often you may find this query useful.

```
CREATE TABLE dbo.poorDesign
(
    poorDesignId int,
    badValue varchar(20)
);
INSERT INTO dbo.poorDesign
VALUES (1,'1,3,56,7,3,6'),
       (2,'22,3'),
       (3,'1');
```

The code just takes the stuff in the WHERE clause of the previous query and moves it into JOIN criteria.

```
SELECT poorDesign.poorDesignId AS betterDesignId,
  SUBSTRING(',' + poorDesign.badValue + ',',I + 1,
    CHARINDEX(',',',' + poorDesign.badValue + ',', I + 1) - I - 1)
       AS betterScalarValue
FROM dbo.poorDesign
  JOIN Tools.Number
    ON I >= 1
    AND I < LEN(',' + poorDesign.badValue + ',') - 1
    AND SUBSTRING(',' + + poorDesign.badValue + ',', I, 1) = ',';
```

This returns the following:

```
betterDesignId          betterScalarValue
--------------          ------------------
1                       1
1                       3
1                       56
1                       7
1                       3
1                       6
2                       22
2                       3
3                       1
```

Ah…that's much better. Each row of the output represents only a single value, not an array of values. As I have said many times throughout the book, SQL works great with atomic values, but try to get individual values out of a single column, and you get ugly code like I have just presented. It is an excellent solution for the

www.it-ebooks.info

problem; in fact, it is the fault of the problem that makes it ugly. Now just create the table using this better design of having one row per scalar value, insert the data, and drop the bad designed table. I won't create the better design, but we do need to clean up the `poorDesign` table with the following, lest someone stumbles upon it and uses it as a good idea:

```
DROP TABLE dbo.poorDesign;
```

Stupid Mathematic Tricks

I want to give you a final, (hopefully) entertaining, and esoteric usage of the sequence table to get your mind working on the possibilities. One of my favorite episodes of *Futurama* is an episode called "Lesser of Two Evils." In this episode, Bender and the Bender look-alike named Flexo start talking and have the following exchange: Bender look-alike named Flexo start talking and have the following exchange (they are both Bender units…Did someone call for a nerd?):

> *Bender: Hey, brobot, what's your serial number?*
> *Flexo: 3370318.*
> *Bender: No way! Mine's 2716057!*
> *Fry (a human): I don't get it.*
> *Bender: We're both expressible as the sum of two cubes!*

So, I figured, the sum of two cubes would be an interesting and pretty easy abstract utilization of the numbers table. "Taxicab" numbers are also mentioned on the ScienceNews.org web site,[1] where the goal is to discover the smallest value that can be expressed as the sum of three cubes in `N` different ways. They are called taxicab numbers because of an old (and pretty darn nerdy) story in which one mathematician remarked to another mathematician that the number 1729 on a taxicab was "dull," to which the other one remarked that it was very interesting, because it was the smallest number that could be expressed as the sum of two cubes. (You can judge your own nerdiness by whether you think: A. This is stupid; B. This is cool; or C. You have done it yourself.)

How hard is the query? It turns out that once you have a sequence table with numbers from 1 to 100,000 or so, you can calculate that `Taxicab(2) = 1729` very easily (and all of the other numbers that are the sum of two cubes too) and the sum of two cubes in three different ways also pretty easily, It took three seconds on my laptop, and that value is `87539319`.

But, instead of calculating the value of each integer cubed (`POWER(i,3)`) for each iteration, you can add a computed column to the table, this time as a `bigint` to give the later calculations room to store the very large intermediate values when you start to multiply the two cube values together. You can do something like the following:

```
ALTER TABLE Tools.Number
  ADD Ipower3 AS CAST(POWER(CAST(I AS bigint),3) AS bigint) PERSISTED;
  --Note that I had to cast I as bigint first to let the power function
  --return a bigint
```

Now, here is the code:

```
DECLARE @level int = 2; --sum of two cubes in @level ways
;WITH cubes AS
(SELECT Ipower3
```

[1] Ivars Peterson, "Taxicab Numbers," www.sciencenews.org/view/generic/id/2948/title/Math_Trek__Taxicab_Numbers.

www.it-ebooks.info

```
FROM  Tools.Number
WHERE I >= 1 AND I < 500) --<<<Vary for performance, and for cheating reasons,
                          --<<<max needed value
SELECT c1.Ipower3 + c2.Ipower3 AS [sum of 2 cubes in @level Ways]
FROM   cubes AS c1
       CROSS JOIN cubes AS c2
WHERE  c1.Ipower3 <= c2.Ipower3 --this gets rid of the "duplicate" value pairs

GROUP  BY (c1.Ipower3 + c2.Ipower3)
HAVING COUNT(*) = @level
ORDER BY [sum of 2 cubes in @level Ways];
```

This will return 559 rows in just a second or two. The first row is 1729, which is the smallest number that is the sum of two cubes in two different ways. OK, breaking this down the cube's CTE, the code is pretty simple.

```
(SELECT  Ipower3
FROM     Tools.Number
WHERE    I >= 1 AND I < 500)
```

This limits the values to a table of cubes, but only the first 499. The next part of the query is a bit more interesting. I sum the two cube values, which I get from cross-joining the CTE to itself.

```
SELECT  c1.Ipower3 + c2.Ipower3 AS [sum of 2 cubes in @level Ways]
FROM    cubes AS c1
            CROSS JOIN cubes AS c2
WHERE   c1.Ipower3 <= c2.Ipower3 --this gets rid of the "duplicate" value pairs
```

The WHERE condition of `c1.i3 <= c2.i3` gets rid of the "duplicate" value pairs since `c1` and `c2` have the same values, so without this, for 1729 you would get the following:

```
c1.i3                 c2.i3
-------               -------
1                     1728
729                   1000
1000                  729
1728                  1
```

These pairs are the same, technically, just in reverses. I don't eliminate equality to allow for the case where both numbers are equal, because they won't be doubled up in this set and if the two cubes were of the same number, it would still be interesting if it were the same as a different sum of two cubes. With the following values:

```
c1.i3                 c2.i3
------                -------
1                     1728
729                   1000
```

Now you can see that 1729 is the sum of two cubes in two different ways. So, lastly, the question of performance must come up. Reading the articles, it is clear that this is not a terribly easy problem to solve as the values get really large. Values for the sum of three cubes are fairly simple. Leaving the sequence values bounded at 500.

www.it-ebooks.info

```
DECLARE @level int = 3; --sum of two cubes in @level ways
;WITH cubes AS
(SELECT Ipower3
FROM Tools.Number
WHERE I >= 1 AND I < 500) --<<< Vary for performance, and for cheating reasons,
                          --<<< max needed value
SELECT c1.Ipower3 + c2.Ipower3 AS [sum of 2 cubes in @level Ways]
FROM   cubes AS c1
       CROSS JOIN cubes AS c2
WHERE  c1.Ipower3 < c2.Ipower3
GROUP  BY (c1.Ipower3 + c2.Ipower3)
HAVING COUNT(*) = @level
ORDER BY [sum of 2 cubes in @level Ways];
```

This returns in around a second:

```
sum of 2 cubes in @level Ways
------------------------------
87539319
119824488
```

Four, however, was a "bit" more challenging. Knowing the answer from the article, I knew I could set a boundary for my numbers using 20000 and get the answer.

```
DECLARE @level int = 4 --sum of two cubes in @level ways
;WITH cubes AS
(SELECT Ipower3
FROM Tools.Number
WHERE I >= 1 and I < 20000) --<<<Vary for performance, and for cheating reasons,
                            --<<<max needed value
SELECT c1.Ipower3 + c2.Ipower3 AS [sum of 2 cubes in @level Ways]
FROM  cubes AS c1
      CROSS JOIN cubes AS c2
WHERE    c1.Ipower3 < c2.Ipower3
GROUP    BY (c1.Ipower3 + c2.Ipower3)
HAVING   COUNT(*) = @level
ORDER BY [sum of 2 cubes in @level Ways];
```

Using this "cheat" of knowing how to tweak the number you need to on my laptop's VM, I was able to calculate that the minimum value of `taxicab(4)` was `6963472309248` (yes, it found only the one) in just 2 hours and 42 seconds on the same VM and laptop mentioned earlier. Clearly, the main value of T-SQL isn't in tricks like this but that using a sequence table can give you the immediate jumping-off point to solve some problems that initially seem difficult.

■ **Caution** Be careful where you try this code for very large values of `@level`. A primary limiting factor is `tempdb` space and you don't want to blow up the tempdb on your production server only to have to explain this query to your manager. Trust me.

www.it-ebooks.info

Calendar Table

It is a common task for a person to want to know how to do groupings and calculations with date values. For example, you might want sales grouped by month, week, year, or any other grouping. You can usually do this using the SQL Server date functions, but often it is costly in performance, and it is always a fairly non-obvious operation. What can truly help with this process is to use a table filled with date values, commonly called a calendar table. Using a calendar table is commonplace in Business Intelligence/OLAP implementations (something that is covered more in Chapter 14), but it certainly can be useful in OLTP databases when you get stuck doing a confusing date range query.

Using the same form of precalculated logic that we applied to the numbers table, we can create a table that contains one row per date. I will set the date as the primary key and then have data related to the date as columns. The following is the basic date table that I currently use. You can extend it as you want to include working days, holidays, special events, and so on, to filter/group by in the same manner as you do with these columns, along with the others I will add later in the section (again, I am working in a copy of `AdventureWorks2012`, bolting on my tools to make life easier).

```
CREATE TABLE Tools.Calendar
(
    DateValue date NOT NULL CONSTRAINT PKtools_calendar PRIMARY KEY,
    DayName varchar(10) NOT NULL,
    MonthName varchar(10) NOT NULL,
    Year varchar(60) NOT NULL,
    Day tinyint NOT NULL,
    DayOfTheYear smallint NOT NULL,
    Month smallint NOT NULL,
    Quarter tinyint NOT NULL
);
```

■ **Note** I wanted to make several of these columns into persisted computed columns, but it turns out that the `DATENAME` and `DATEPART` functions we will use to load the data were not deterministic functions due to how they have to work with regionalization, so we will store the values manually. In SQL Server 2012, there is an additional way to format date data using `FORMAT` (also non-deterministic), which allows for specification of regionalization information. I didn't change my example to use `FORMAT` as `DATEPART` and `DATENAME` are perfectly valid and a bit more readable methods of extracting date information from a date value, but `FORMAT` would allow you to reference multiple cultures in a single statement if you needed it.

The question of normalization might come up again at this point, and it is a valid question. Since these values we have created can be calculated (and quite easily in most case), why do this? Isn't this denormalization? There are a few ways to look at it, but I would generally say that it isn't. Each row represents a day in time, and each of the columns is functionally dependent on the date value. The fact is, the functions have to look up or calculate the date attributes in some manner, so you are actually saving that lookup cost, however minimal it might be.) When you have to filter on date criteria, the benefit of the normalized design can be quite obvious. Later in the section we will extend the table to add more interesting, company-based values, which certainly are normalized in the context of a date.

Note that the name calendar is not truly consistent with our names representing a single row, at least not in a natural way, but the other names that we might use (`date`, `dateValue`, etc.) all sound either forced or hokey. Since the table represents a calendar and a calendar item generally would logically represent a day, I went with it.

www.it-ebooks.info

A stretch perhaps but naming is pretty difficult at times, especially when working in a namespace that the real world and SQL Server have so many established names.

The next step is to load the table with values, which is pretty much a straightforward task using the `Numbers` table that we just finished creating in the previous section. Using the `datename` and `datepart` functions and a few simple case expressions, you load the different values. I will make use of many of the functions in the examples, but most are very easy to understand.

```
WITH dates (newDateValue) AS (
    SELECT DATEADD(day,I,'17530101') AS newDateValue
    FROM Tools.Number
)
INSERT Tools.Calendar
    (DateValue ,DayName
    ,MonthName ,Year ,Day
    ,DayOfTheYear ,Month ,Quarter
)
SELECT
    dates.newDateValue as DateValue,
    DATENAME (dw,dates.newDateValue) As DayName,
    DATENAME (mm,dates.newDateValue) AS MonthName,
    DATENAME (yy,dates.newDateValue) AS Year,
    DATEPART(day,dates.newDateValue) AS Day,
    DATEPART(dy,dates.newDateValue) AS DayOfTheYear,
    DATEPART(m,dates.newDateValue) AS Month,
    DATEPART(qq,dates.newDateValue) AS Quarter
FROM  dates
WHERE dates.newDateValue BETWEEN '20000101' AND '20130101' --set the date range
ORDER BY DateValue;
```

Just like the numbers table, there are several commonly useful ways to use the calendar table. The first I will demonstrate is general grouping types of queries, and the second is calculating ranges. As an example of grouping, say you want to know how many sales had been made during each year in `Sales.SalesOrderHeader` in the `AdventureWorks2012` database. This is why there is a `year` column in the table:

```
SELECT Calendar.Year, COUNT(*) as OrderCount
FROM /*Adventureworks2012.*/ Sales.SalesOrderHeader
    JOIN Tools.Calendar
          --note, the cast here could be a real performance killer
          --consider using date columns where possible
            ON CAST(SalesOrderHeader.OrderDate as date) = Calendar.DateValue
GROUP BY Calendar.Year
ORDER BY Calendar.Year;
```

This returns the following:

```
Year                    OrderCount
----                    -----------
2005                    1379
2006                    3692
2007                    12443
2008                    13951
```


www.it-ebooks.info

The beauty of the calendar table is that you can easily group values that are not that simple to compute in a really obvious manner them, all while using natural relational coding style/technique. For example: to count the sales on Tuesdays and Thursdays?

```
SELECT Calendar.DayName, COUNT(*) as OrderCount
FROM /*Adventureworks2012.*/ Sales.SalesOrderHeader
    JOIN Tools.Calendar
          --note, the cast here could be a real performance killer
          --consider using date columns where
    ON CAST(SalesOrderHeader.OrderDate AS date) = Calendar.DateValue
WHERE Calendar.DayName IN ('Tuesday','Thursday')
GROUP BY Calendar.DayName
ORDER BY Calendar.DayName;
```

This returns the following:

```
DayName                 OrderCount
-------                 ----------
Thursday                4875
Tuesday                 4483
```

■ **Tip** In many tables where only the date is used in computations like this, I will add a computed column that has the date only. For example, in the `SalesOrderHeader` table in `AdventureWorks2012`, I would have named `OrderDate` `OrderTime` and added a computed column named `OrderDate` defined as `CAST(SalesOrderHeader.OrderDate AS date)` that could have statistics and be indexed for faster operation.

OK, I see you are possibly still skeptical. What if I throw in the first Tuesday after the second Wednesday? How? Well, it is really a simple matter of building the set step by step using CTEs to build the set that you need.

```
;WITH onlyWednesdays AS --get all Wednesdays
(
  SELECT *,
    ROW_NUMBER() OVER (PARTITION BY Calendar.Year, Calendar.Month
                       ORDER BY Calendar.Day) AS wedRowNbr
  FROM Tools.Calendar
  WHERE DayName = 'Wednesday'
),
secondWednesdays AS --limit to second Wednesdays of the month
(
  SELECT *
  FROM  onlyWednesdays
  WHERE wedRowNbr = 2
)
,finallyTuesdays AS --finally limit to the Tuesdays after the second wed
(
```


www.it-ebooks.info

```
  SELECT Calendar.*,
    ROW_NUMBER() OVER (PARTITION BY Calendar.Year, Calendar.Month
                       ORDER by Calendar.Day) AS rowNbr
  FROM secondWednesdays
       JOIN Tools.Calendar
         ON secondWednesdays.Year = Calendar.Year
           AND secondWednesdays.Month = Calendar.Month
  WHERE Calendar.DayName = 'Tuesday'
    AND Calendar.Day > secondWednesdays.Day
)
--and in the final query, just get the one month
SELECT Year, MonthName, Day
FROM   finallyTuesdays
WHERE  Year = 2012
       AND rowNbr = 1;
```

This returns the following set of values that appear to be the result of some advanced calculations, but rather are the results of a fairly simple query (to anyone in Chapter 12 of this book, for sure!):

```
Year        MonthName         Day
----        ---------         ----
2012        January           17
2012        February          14
2012        March             20
2012        April             17
2012        May               15
2012        June              19
2012        July              17
2012        August            14
2012        September         18
2012        October           16
2012        November          20
2012        December          18
```

Now, utilizing another CTE, you could use this to join to the `Sales.SalesOrderHeader` table and find out sales on the first Tuesday after the second Wednesday. And that is exactly what I would do if I was being fed requirements by a madman (or perhaps they prefer the term is marketing analyst?) and would be done in no time. However, if this date became a corporate standard desired date, , I would add a column to the calendar table (maybe called `FirstTuesdayAfterSecondWednesdayFlag`) and set it to 1 for every date that it matches (and in the one month where that day was a holiday and they wanted it to be the Thursday after the second Wednesday after…well, the change would be a simple update.

The realistic application of this is a company sale that they routinely start on a given day every month, so the report needs to know how sales were for four days after. So, perhaps the column would be `BigSaleDaysFlag`, or whatever works. The goal is the same as back in Chapter 7 when we talked about data driven design. Store data in columns; rules in code. A new type of day is just a new column and calculation that every user now has access to and doesn't have to know the calculation that was used to create the value.

www.it-ebooks.info

In most standard calendar table utilizations, it will be common to have, at a minimum, the following generic events/time ranges:

```
--Saturday or Sunday set to 1, else 0
WeekendFlag bit NOT NULL,

FiscalYear smallint NOT NULL,
FiscalMonth tinyint NULL,
FiscalQuarter tinyint NOT NULL
```

Almost every company has a fiscal year that it uses for its business calendar. This technique allows you to treat the fiscal time periods more or less like regular calendar dates in your code, with no modification at all. I will demonstrate this after we load the data in just a few paragraphs.

This covers the basics; now I'll discuss one last thing you can add to the calendar table to solve the problem that really shifts this into "must-have" territory: floating windows of time.

■ **Note** This is actually one of the few techniques that I created on my own. I am not actually claiming that I am the only person to ever do this, but I have not read about it anywhere else, and I built this myself when trying to solve a particular type of problem using the system functions to no avail.

The goal for the floating windows of time was to add a relative positioning value to the table so that you could easily get N time periods from a point in time. Years are already contiguous, increasing numbers, so it is easy to do math with them. But it is not particularly comfortable to do math with months. I found myself often having to get the past 12 months of activity, sometimes including the current month and sometimes not. Doing math that wraps around a 12-month calendar was a pain, so to the calendar, I add the following columns that I will load with increasing values with no gaps:

```
RelativeDayCount int NOT NULL,
RelativeWeekCount int NOT NULL,
RelativeMonthCount int NOT NULL
```

Using these columns I will store sequence numbers that start at an arbitrary point in time. I will use `'20000101'` here, but it is really unimportant, and negative values are not a problem either. You should never refer to the value itself, just the value's relative position to some point in time you choose. And days are numbered from that point (negative before, positive before), months, and again weeks. This returns the following "final" calendar table:

```
DROP TABLE Tools.Calendar
GO
CREATE TABLE Tools.Calendar
(
    DateValue date NOT NULL CONSTRAINT PKtools_calendar PRIMARY KEY,
    DayName varchar(10) NOT NULL,
    MonthName varchar(10) NOT NULL,
    Year varchar(60) NOT NULL,
    Day tinyint NOT NULL,
    DayOfTheYear smallint NOT NULL,
    Month smallint NOT NULL,
    Quarter tinyint NOT NULL,
    WeekendFlag bit NOT NULL,
```


www.it-ebooks.info

```
    --start of fiscal year configurable in the load process, currently
    --only supports fiscal months that match the calendar months.
    FiscalYear smallint NOT NULL,
    FiscalMonth tinyint NULL,
    FiscalQuarter tinyint NOT NULL,

    --used to give relative positioning, such as the previous 10 months
    --which can be annoying due to month boundaries
    RelativeDayCount int NOT NULL,
    RelativeWeekCount int NOT NULL,
    RelativeMonthCount int NOT NULL
)
```

Last, I will reload the table with the following code:

```
;WITH dates (newDateValue) AS (
    SELECT DATEADD(day,I,'17530101') AS newDateValue
    FROM Tools.Number
)
INSERT Tools.Calendar
    (DateValue ,DayName
    ,MonthName ,Year ,Day
    ,DayOfTheYear ,Month ,Quarter
    ,WeekendFlag ,FiscalYear ,FiscalMonth
    ,FiscalQuarter ,RelativeDayCount,RelativeWeekCount
    ,RelativeMonthCount)
SELECT
    dates.newDateValue AS DateValue,
    DATENAME (dw,dates.newDateValue) AS DayName,
    DATENAME (mm,dates.newDateValue) AS MonthName,
    DATENAME (yy,dates.newDateValue) AS Year,
    DATEPART(day,dates.newDateValue) AS Day,
    DATEPART(dy,dates.newDateValue) AS DayOfTheYear,
    DATEPART(m,dates.newDateValue) AS Month,
    CASE
        WHEN MONTH( dates.newDateValue) <= 3 THEN 1
        WHEN MONTH( dates.newDateValue) <= 6 THEN 2
        When MONTH( dates.newDateValue) <= 9 THEN 3
    ELSE 4 END AS quarter,

    CASE WHEN DATENAME (dw,dates.newDateValue) IN ('Saturday','Sunday')
        THEN 1
        ELSE 0
    END AS weekendFlag,
    -----------------------------------------------
    --the next three blocks assume a fiscal year starting in July.
    --change if your fiscal periods are different
    -----------------------------------------------
    CASE
        WHEN MONTH(dates.newDateValue) <= 6
        THEN YEAR(dates.newDateValue)
        ELSE YEAR (dates.newDateValue) + 1
```


www.it-ebooks.info

```
    END AS fiscalYear,

    CASE
        WHEN MONTH(dates.newDateValue) <= 6
        THEN MONTH(dates.newDateValue) + 6
        ELSE MONTH(dates.newDateValue) - 6
    END AS fiscalMonth,

    CASE
        WHEN MONTH(dates.newDateValue) <= 3 then 3
        WHEN MONTH(dates.newDateValue) <= 6 then 4
        WHEN MONTH(dates.newDateValue) <= 9 then 1
    ELSE 2 END AS fiscalQuarter,
    -----------------------------------------------
    --end of fiscal quarter = july
    -----------------------------------------------

    --these values can be anything, as long as they
    --provide contiguous values on year, month, and week boundaries
    DATEDIFF(day,'20000101',dates.newDateValue) AS RelativeDayCount,
    DATEDIFF(week,'20000101',dates.newDateValue) AS RelativeWeekCount,
    DATEDIFF(month,'20000101',dates.newDateValue) AS RelativeMonthCount

FROM  dates

WHERE dates.newDateValue BETWEEN '20000101' AND '20130101'; --set the date range
```

Now we can build a query to get only weekends, grouped by fiscalYear as follows:

```
SELECT Calendar.FiscalYear, COUNT(*) AS OrderCount
FROM /*Adventureworks2012.*/ Sales.SalesOrderHeader
    JOIN Tools.Calendar
        --note, the cast here could be a real performance killer
        --consider using a persisted calculated column here
      ON CAST(SalesOrderHeader.OrderDate as date) = Calendar.DateValue
WHERE WeekendFlag = 1
GROUP BY Calendar.FiscalYear
ORDER BY Calendar.FiscalYear;
```

This returns the following:

```
FiscalYear      OrderCount
----------      ----------
2006            734
2007            1114
2008            6855
2009            234
```

To demonstrate the floating windows of time using the Relative_____Count columns, consider that you want to count the sales for the previous two weeks. It's not impossible to do this using the date functions perhaps, but it's simple to do with a calendar table:

```
DECLARE @interestingDate date = '20120509';
```


www.it-ebooks.info

```
SELECT Calendar.DateValue as PreviousTwoWeeks, CurrentDate.DateValue AS Today,
    Calendar.RelativeWeekCount
FROM Tools.Calendar
    JOIN (SELECT *
          FROM Tools.Calendar
          WHERE DateValue = @interestingDate) AS CurrentDate
    ON Calendar.RelativeWeekCount < (CurrentDate.RelativeWeekCount)
        AND Calendar.RelativeWeekCount >=
            (CurrentDate.RelativeWeekCount -2);
```

This returns the following:

```
PreviousTwoWeeks       Today              RelativeWeekCount
-----------------      ----------         ------------------
2012-04-22             2012-05-09         643
2012-04-23             2012-05-09         643
2012-04-24             2012-05-09         643
2012-04-25             2012-05-09         643
2012-04-26             2012-05-09         643
2012-04-27             2012-05-09         643
2012-04-28             2012-05-09         643
2012-04-29             2012-05-09         644
2012-04-30             2012-05-09         644
2012-05-01             2012-05-09         644
2012-05-02             2012-05-09         644
2012-05-03             2012-05-09         644
2012-05-04             2012-05-09         644
2012-05-05             2012-05-09         644
```

From this, we can see that the previous two weeks start on Sunday (04/22/2012 and 04/29/2012) and end on Saturday (04/28/2012 and 05/05/2012). The dates 05/06/2012 to 05/09/2012 are not included because that is this week (relative to the variable value). The basics of the query is simply to take a derived table that fetches the calendar row for the "interesting" date and then join that to the full calendar table using a range of dates in the join. In the previous week's example, I used the following:

```
Calendar.RelativeWeekCount < (CurrentDate.RelativeWeekCount)
    AND Calendar.RelativeWeekCount >= (CurrentDate.RelativeWeekCount -2)
```

This was because I wanted the weeks that are previous to the current week and weeks two weeks back. Weeks aren't the sweet spot of this technique exactly, because weeks are of fixed length (but they are easier to get a full result set in print, since I don't get paid by the page). Now, join the values to your sales table, and you can see sales for the past two weeks. Want to see it broken down by week? Use relative week count. Use the following if you wanted the previous 12 months:

```
DECLARE @interestingDate date = '20120509'

SELECT MIN(Calendar.DateValue) AS MinDate, MAX(Calendar.DateValue) AS MaxDate
FROM Tools.Calendar
      JOIN (SELECT *
            FROM Tools.Calendar
```


www.it-ebooks.info

```
            WHERE DateValue = @interestingDate) AS CurrentDate
    ON Calendar.RelativeMonthCount < (CurrentDate.RelativeMonthCount)
        AND Calendar.RelativeMonthCount >=
            (CurrentDate.RelativeMonthCount -12);
```

This returns the following:

```
MinDate          MaxDate
----------       ----------
2011-05-01       2012-04-30
```

This query includes all earlier months, but not the one loaded into `@interestingDate`. Notice that the results did not include the current month. Change the < to <=, and it will include the current month. If you want to get a 24-month window, get the 12 months plus or minus the current month, as follows:

```
ON Calendar.RelativeMonthCount < (CurrentDate.RelativeMonthCount + 12)
    AND Calendar.RelativeMonthCount >=
        (CurrentDate.RelativeMonthCount -12)
```

Using that `ON` clause results in the following:

```
MinDate          MaxDate
----------       ----------
2007-03-01       2009-02-28
```

Now you can use these dates in other criteria, either by assigning these values to a variable or (if you are one of the cool kids) by using the tables in a join to other tables. As a real example, let's hop in the Wabac machine and look at sales in the `Adventureworks2012` database around September 27, 2008. (Keep in mind that the sample databases could change in the future to get more up-to-date data, but the 2012 RTM version had dates that were the same as the 2008 and the 2008 R2 editions of `AdventureWorks`.)

```
DECLARE @interestingDate date = '20080927';

SELECT Calendar.Year, Calendar.Month, COUNT(*) AS OrderCount
FROM /*Adventureworks2012.*/ Sales.SalesOrderHeader
      JOIN Tools.Calendar
         JOIN (SELECT *
               FROM Tools.Calendar
               WHERE DateValue = @interestingDate) AS CurrentDate
            ON Calendar.RelativeMonthCount <=
              (CurrentDate.RelativeMonthCount )
            AND Calendar.RelativeMonthCount >=
              (CurrentDate.RelativeMonthCount -10)
      ON cast(SalesOrderHeader.ShipDate as date)= Calendar.DateValue
GROUP BY Calendar.Year, Calendar.Month
ORDER BY Calendar.Year, Calendar.Month;
```


www.it-ebooks.info

This query will give you items that shipped in the previous ten months and in August, and group them by month, as follows:

```
Year        Month       OrderCount
-----       ------      -----------
2007        11          1825
2007        12          2228
2008        1           1998
2008        2           2034
2008        3           2100
2008        4           2058
2008        5           2395
2008        6           2360
2008        7           1271
2008        8           211
```

I included the current month by changing the first condition to <=. So, you should be able to see that the calendar table and sequence table are two excellent tables to add to almost any database. They give you the ability to take a functional problem like getting the last day of the month or the previous time periods and turn it into a relational question that SQL Server can chew up and spit out using the relational techniques it is built for.

Keep in mind too that creating more fiscal calendars, reporting calendars, corporate holiday calendars, sales calendars, and so forth, is easily done by adding more columns to this very same calendar table. For instance, you may have multiple business units, each with their own fiscal calendar. With the one normal calendar, you can add any variation of the fiscal calendar multiple times. It's easy to just add those columns, since they all relate back to a normal calendar date.

One last thing to note is that the calendar table needs to be maintained. The insert statement we wrote earlier can be changed slightly so that it can be run to extend the number of rows as time passes. Perhaps an Agent job ran on the first day of the year to add another set of rows, or I have also used a process to keep the calendar just a few weeks in the future for a payroll system. The rules changed on occasion, so the new rows were added just a few weeks in advance based on whatever rules were then in effect.

Utility Objects

As you acquire more and more experience over the years (which is loosely synonymous with the many failures you will encounter that teach you lessons), you will undoubtedly end up with a toolbox full of various tools that you find you use quite frequently. Some of these tools will be used in an ad-hoc manner when you are dealing with the same problems over and over again. I personally like to simply keep a set of files/projects stored on my dropbox/skydrive folders for easy accessibility when I am working, writing, or just poking around at SQL Server to see what I can see at the office, home, or even at a client site.

Beyond the type of utilities that are used infrequently, the toolbox will contain a rather large set of tools that become part of the regular processing of the system. For these types of tools the logistics of their use can pose a problem. How do you make sure that they are usable by every database, as needed, and allow for the differences that will occur in every system (and keep your databases portable so that you can move them to a new server with the least amount of pain? In the old days of the twentieth century, we would make a system object in the `master` database that was prefixed with sp_ (and marked it as a system object) and diligently maintain all servers as equal as possible While this is still possible, we found that too often if we wanted to include a change to the object, we couldn't do it because with N systems using an object, we usually found N different ways an object was used. Then you would have abnormal procedures spread around the enterprise with names that included a database

www.it-ebooks.info

that might be on a single server (and over time, didn't even exist anymore). As time passed, we stopped using master and started adding a utility database which was managed on every server. Ideally they were identical, but over time, the DBA and or Developers used the database for "temp" usages that again becomes unwieldy and decidedly not temporary.

What seems to be the best answer is to create a schema on each database that contains the tools that that database actually uses. By putting the objects in each database individually, it is easier to decide whether additions to an object should be a new object that is specific to the single database or if it was simply a new version of an existing object. If it is just a new version, the systems that were using an object would take the new change as an upgrade after due testing. In either case, each database becomes more stand-alone so that moving from server A to server B means a lot less preparation on server B trying to meet the needs of server B. A few SQL Agent jobs may need to be created to execute.

For examples, I chose a few of the types of utilities that I find useful in my databases I design. I will present one example in each, though this is only just a start. We will look at solutions in:

- *Monitoring tools*: Tools to watch what is going on with the database such as row counts, file sizes, and so forth.
- *Extended DDL utilities*: Tools used to make changes to the structure of the database, usually to remove keys or indexes for load processes, usually to do multiple DDL calls where SQL Server's DDL would require multiple calls.

Monitoring Objects

Keeping an eye on the database usage is a very common need for the database administrator. A very typical question from upper management is to find out how much a system is used, and particularly how much the database has grown over time. With a little bit of planning it is easy to be prepared for these questions and others by doing a bit of monitoring.

In the example, I will build a table that will capture the row counts from all of the tables in the database (other than the sys schema), but running a stored procedure. It is set up to use the `sys.partitions` catalog view, because it gives a "good enough" count of the rows in all of the tables. If it is important to get extremely precise rowcounts, you could use a cursor and do a select `COUNT(*)` from each table in the database as well.

For monitoring data, I will create a `Monitor` schema. This I will not give rights to anyone by default, as the monitor schema will generally be for the DBA to get a feel for system growth.

```
CREATE SCHEMA Monitor;
```

Next I will create a table, with the grain of the data being at a daily level. The procedure will be created to allow you to capture rowcounts more than once a day if needed. Note too that the `ObjectType` column can have more than just tables, since it might be interesting to see if indexed views are also growing in size as well. I will include only clustered or heap structures so that we get only the table structures.

```
CREATE TABLE Monitor.TableRowCount
(
    SchemaName sysname NOT NULL,
    TableName sysname NOT NULL,
    CaptureDate date NOT NULL,
    Rows integer NOT NULL,
    ObjectType sysname NOT NULL,
    Constraint PKMonitor_TableRowCount PRIMARY KEY (SchemaName, TableName, CaptureDate)
);
```


www.it-ebooks.info

Then the following procedure will be scheduled to run daily:

```
CREATE PROCEDURE Monitor.TableRowCount$captureRowcounts
AS
-- ------------------------------------------------------------------
-- Monitor the row counts of all tables in the database on a daily basis
-- Error handling not included for example clarity
--
-- NOTE: This code expects the Monitor.TableRowCount to be in the same db as the
-- tables being monitored. Rework would be needed if this is not a possibility
--
-- 2012 Louis Davidson – drsql@hotmail.com – drsql.org
-- ------------------------------------------------------------------

-- The CTE is used to set upthe set of rows to put into the Monitor.TableRowCount table
WITH CurrentRowcount AS (
SELECT  OBJECT_SCHEMA_NAME(partitions.object_id) AS SchemaName,
        OBJECT_NAME(partitions.object_id) AS TableName,
        CAST(getdate() AS date) AS CaptureDate,
        SUM(rows) AS Rows,
        objects.type_desc AS ObjectType
FROM    sys.partitions
                JOIN sys.objects
                        ON partitions.object_id = objects.object_id
WHERE   index_id IN (0,1) --Heap 0 or Clustered 1 “indexes”
AND     OBJECT_SCHEMA_NAME(partitions.object_id) NOT IN ('sys')
--the GROUP BY handles partitioned tables with > 1 partition
GROUP BY partitions.object_id, objects.type_desc)

--MERGE allows this procedure to be run > 1 a day without concern, it will update if the row
--for the day exists
MERGE   Monitor.TableRowCount
USING   (SELECT SchemaName, TableName, CaptureDate, Rows, ObjectType
        FROM CurrentRowcount) AS Source
                ON (Source.SchemaName = TableRowCount.SchemaName
                    AND Source.TableName = TableRowCount.TableName
                    AND Source.CaptureDate = TableRowCount.CaptureDate)
WHEN MATCHED THEN
        UPDATE SET Rows = Source.Rows
WHEN NOT MATCHED THEN
        INSERT (SchemaName, TableName, CaptureDate, Rows, ObjectType)
        VALUES (Source.SchemaName, Source.TableName, Source.CaptureDate,
                Source.Rows, Source.ObjectType);
GO
```

Now, you execute the following procedure and check the results for the `HumanResources` schema in `AdventureWorks2012`, where we have been working:

```
EXEC Monitor.TableRowCount$captureRowcounts;

SELECT *
FROM  Monitor.TableRowCount
WHERE SchemaName = 'HumanResources'
ORDER BY SchemaName, TableName;
```


www.it-ebooks.info

Then (assuming you are still in the pristine version of the AddIn database we are working on), the output of this batch will be as follows:

```
SchemaName          TableName                 CaptureDate  Rows  ObjectType
------------------- ------------------------- -----------  ----  -----------
HumanResources      Department                2012-02-25   16    USER_TABLE
HumanResources      Employee                  2012-02-25   290   USER_TABLE
HumanResources      EmployeeDepartmentHistory 2012-02-25   296   USER_TABLE
HumanResources      EmployeePayHistory        2012-02-25   316   USER_TABLE
HumanResources      JobCandidate              2012-02-25   13    USER_TABLE
HumanResources      Shift                     2012-02-25   3     USER_TABLE
```

If you look at all of the rows in the table for the Monitor schema, you will see that they were 0, since it was checked before we added these rows. Run it again and you will notice that there is an increase of rows in the Monitor.TableRowCount table, notably the rows we just added. I tend to capture the rowcount of all tables in the Monitor and Tools database as well. In many cases, I will then add a procedure to check for abnormal growth of rows in a table. For example, if the calendar table changes rows (up or down), there could easily be an issue, since this table will generally grow once, at the end of the year. You might also write a query to make sure that the monitoring table rows are increasing and alert the admin if not.

Extended DDL Utilities

In a high number of the systems I work with, data is constantly being moved around, sometimes a very large amount. I almost always make sure that there are relationships, check constraints, unique constraints, etc. This is a way to make sure that the data that is loaded meets the needed quality standards that the user demands.

However, constraints can really slow down the loading of data so quite often the loading program (like SSIS) disables your constraints to make it load the data faster. Unfortunately, it doesn't re-enable the constraints after it is done loading the data. So I created the following procedure to re-enable the constraints in some or all of the tables in the database. I will put the procedure in a Utility schema and restrict access to only sysadmin users (even going so far as to use a DENY permission for non-sysadmin access).

```
CREATE SCHEMA Utility;
GO
CREATE PROCEDURE Utility.Constraints$ResetEnableAndTrustedStatus
(
        @table_name sysname = '%',
        @table_schema sysname = '%',
        @doForeignKeyFlag  bit = 1,
        @doCheckFlag bit = 1
) as
-- ------------------------------------------------------------
-- Enables disabled foreign key and check constraints, and sets
-- trusted status so optimizer can use them
--
-- NOTE: This code expects the Monitor.TableRowCount to be in the same db as the
-- tables being monitored. Rework would be needed if this is not a possibility
--
-- 2012 Louis Davidson - drsql@hotmail.com - drsql.org
-- ------------------------------------------------------------
```


www.it-ebooks.info

```
BEGIN

    SET NOCOUNT ON;
    DECLARE @statements cursor; --use to loop through constraints to execute one
                                --constraint for individual DDL calls
SET @statements = cursor for
    WITH FKandCHK AS (SELECT OBJECT_SCHEMA_NAME(parent_object_id) AS schemaName,
                             OBJECT_NAME(parent_object_id) AS tableName,
                             NAME AS constraintName, Type_desc AS constraintType,
                             is_disabled AS DisabledFlag,
                             (is_not_trusted + 1) % 2 AS TrustedFlag
                      FROM sys.foreign_keys
                      UNION ALL
                      SELECT OBJECT_SCHEMA_NAME(parent_object_id) AS schemaName,
                             OBJECT_NAME(parent_object_id) AS tableName,
                             NAME AS constraintName, Type_desc AS constraintType,
                             is_disabled AS DisabledFlag,
                             (is_not_trusted + 1) % 2 AS TrustedFlag
                      FROM   sys.check_constraints )
SELECT schemaName, tableName, constraintName, constraintType,
       DisabledFlag, TrustedFlag
FROM   FKandCHK
WHERE  (TrustedFlag = 0 OR DisabledFlag = 1)
AND    ((constraintType = 'FOREIGN_KEY_CONSTRAINT' AND @doForeignKeyFlag = 1)
       OR (constraintType = 'CHECK_CONSTRAINT' AND @doCheckFlag = 1))
AND    schemaName LIKE @table_Schema
AND    tableName LIKE @table_Name;
OPEN @statements;

DECLARE @statement varchar(1000), @schemaName sysname,
        @tableName sysname, @constraintName sysname,
        @constraintType sysname,@disabledFlag bit, @trustedFlag bit;
WHILE 1=1
BEGIN
FETCH FROM @statements INTO @schemaName, @tableName, @constraintName,
        @constraintType, @disabledFlag, @trustedFlag;
IF @@FETCH_STATUS <> 0
    BREAK;

BEGIN TRY -- will output an error if it occurs but will keep on going
    --so other constraints will be adjusted
    IF @constraintType = 'CHECK_CONSTRAINT'
    SELECT @statement = 'ALTER TABLE ' + @schemaName + '.'
        + @tableName + ' WITH CHECK CHECK CONSTRAINT '
        + @constraintName;
    ELSE IF @constraintType = 'FOREIGN_KEY_CONSTRAINT'
    SELECT @statement = 'ALTER TABLE ' + @schemaName + '.'
        + @tableName + ' WITH CHECK CHECK CONSTRAINT '
        + @constraintName;
    EXEC (@statement);
END TRY
BEGIN CATCH --output statement that was executed along with the error number
```


www.it-ebooks.info

```
        SELECT 'Error occurred: ' + CAST(ERROR_NUMBER() AS varchar(10))+ ':' +
        ERROR_MESSAGE() + CHAR(13) + CHAR(10) + 'Statement executed: ' +
        @statement;
    END CATCH
    END;
    END;
GO
```

I have several more of these available in the downloads of my web site (`drsql.org`) to manage (and mostly drop) all types of objects in bulk (constraints, indexes, etc) for when you need to remove the constraints from a table, often in an attempt to update the structure. I keep a model database/structure snapshot of how the database should look, and then I can remove anything I need to and simply add it back using a comparison tool.

Logging Objects

In many systems, you will find you need to watch the activities that are occurring in the database. Back in Chapter 7 we implemented an audit trail using triggers, and we chose between using the source table schema or some form of utility schema. In this section, the types of logging we will want to do are a more generic form of logging that is more for the DBAs to see what errors have been occurring.

As an example, one thing that we often may want to log is errors. A common goal these days is to make sure that no errors occur at the database level. Logging any errors that occur to a table can help to see where you have recurring issues. Probably the most interesting way this has ever helped me in my systems was once when a programmer had simply ignored all return values from SQL calls. So a large number of calls to the system were failing, but the client never realized it. (In Appendix B, I will employ this functionality in the trigger templates that I provide).

The `Utility.ErrorLog$insert` procedure is used to log the errors that occur in a table, to give you a history of errors that have occurred. I do this because, in almost every case, an error that occurs in a trigger is a bad thing. The fact that the client sends data that might cause the trigger to fail should be fixed and treated as a bug. In stored procedures, this may or may not be the case, as stored procedures can be written to do things that may work, or may fail in some situations. This is a very broad statement, and in some cases may not be true, so you can adjust the code as fits your desires.

The DML for the table is as follows:

```
CREATE TABLE Utility.ErrorLog(
    ErrorLogId int NOT NULL IDENTITY CONSTRAINT PKErrorLog PRIMARY KEY,
          Number int NOT NULL,
    Location sysname NOT NULL,
    Message varchar(4000) NOT NULL,
    LogTime datetime2(3) NULL
        CONSTRAINT DFLTErrorLog_error_date DEFAULT (SYSDATETIME()),
    ServerPrincipal sysname NOT NULL
        --use ORIGINAL_LOGIN to capture the user name of the actual user
        --not a user they have impersonated
          CONSTRAINT DFLTErrorLog_error_user_name DEFAULT (ORIGINAL_LOGIN())
);
```

Then we create the following procedure, which can be coded into other procedures and trigger whenever you need to log that an error occurred:

```
CREATE PROCEDURE Utility.ErrorLog$Insert
(
    @ERROR_NUMBER int,
```


www.it-ebooks.info

```
    @ERROR_LOCATION sysname,
    @ERROR_MESSAGE varchar(4000)
) AS
-- ------------------------------------------------------------
-- Writes a row to the error log. If an error occurs in the call (such as a NULL value)
-- It writes a row to the error table. If that call fails an error will be returned
--
-- 2012 Louis Davidson - drsql@hotmail.com - drsql.org
-- ------------------------------------------------------------

 BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
        INSERT INTO Utility.ErrorLog(Number, Location, Message)
        SELECT @ERROR_NUMBER,COALESCE(@ERROR_LOCATION,'No Object'),@ERROR_MESSAGE;
    END TRY
    BEGIN CATCH
        INSERT INTO Utility.ErrorLog(Number, Location, Message)
        VALUES (-100, 'Utility.ErrorLog$insert',
        'An invalid call was made to the error log procedure ' +
            ERROR_MESSAGE());
    END CATCH
END;
```

Then we test the error handler with a simple test case, as follows:

```
--test the error block we will use
BEGIN TRY
    THROW 50000,'Test error',16;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0
      ROLLBACK TRANSACTION;

    --[Error logging section]
      DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
            @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
            @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE();
    EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;

    THROW; --will halt the batch or be caught by the caller's catch block
END CATCH
```

This returns the error we threw.

```
Msg 50000, Level 16, State 16, Line 3
Test error
```

And checking the ErrorLog, you can see that the error is logged. (Say that three times fast. I dare you.)

```
SELECT *
FROM   Utility.ErrorLog;
```


www.it-ebooks.info

This returns the following:

```
ErrorLogId     Number      Location      Message       LogTime          ServerPrincipal
---------------  ----------  ---------------  ----------  ----------------  ------------------------------
50000          No Object   Test error    2012-02-01    22:18:25.615     DENALI-PC\AlienDrsql
```

This basic error logging procedure can make it much easier to understand what has gone wrong when a user has an error. Expand your own system to meet your organization's needs, but having an audit trail will prove invaluable when you find out that certain types of errors have been going on for weeks and your users "assumed" you knew about it!

The only real downside to logging in this manner is transactions. You can log all you want, but if the log procedure is called in a transaction, and that transaction is rolled back, the log row will also be rolled back. To write to a log that isn't affected by transactions, you can use the `xp_logevent` extended stored procedure in the error handler to write to the Windows Event Log. Using this method can be handy if you have deeply nested errors, in which all the `dbo.ErrorLog` rows get rolled back due to external transactions.

Other Possibilities…

In the end, it is really going to be up to your database's particular needs to determine exactly what you may need to put into every (or certainly greater than one) database in your organization. In practice, I tend to see some utilities positioned in a database that are used on the server, for backups, generic index maintenance, and other sorts of generic maintenance.

In my practice, anytime the application and/or any database code references code or data from within the database, everything gets created in the database with the other code. Some other types of concepts that I have seen added to the database to enhance the application are:

- *Functions that aren't implemented in SQL Server*: User-defined functions have plenty of pros and cons when used in T-SQL statements (particularly in a `WHERE` clause) but they definitely have their place to wrap common, especially complex, functionality into a neat package. For example, consider the functionality we covered in the earlier section entitled: "Separating Comma Delimited Items." We might create a function `Tools.String$Split (@ValueToSplit)` that encapsulates the string splitting function for reuse.
- *Security*: Microsoft provides a lot of security oriented system functions for you to determine who the user is, but sometimes it isn't enough. For some systems, I have employed a function in a schema named security to allow for overriding the system context information when needed. The goal being that whenever you need to know what user is making changes to your system, you can call `Security.userName$get()` function and know that it is doing the heavy lifting, either looking to `ORIGINAL_LOGIN()` system function or using a custom built security system.
- *Reference/demographic information*: It is very common for systems to need the city, state, zip/postal code, phone number, and/or country information of customers/clients. Several companies (including the United States Postal Service) publish the domain data that defines the regions and format of this data, but in every case I have seen, they provide the data in arcane formats that are difficult to work with. So we create a demographics database with a schema (we use `reference` for ours) that we can duplicate in any system that needs this sort of data so we have to load it once from the foreign format, and in the expected format everywhere else it is needed making it easier to share around the enterprise.

www.it-ebooks.info

As we said, the sky is the limit; use your imagination as to how you can build up your code base in a way that is reusable and also will enhance your application's needs.

Summary

In most chapters we end with a section covering best practices, but in this chapter there really is only one primary best practice: keep your code encapsulated within the bounds of each database. Even if you use the same code in every database and it is the same now, it may not remain that way forever. The biggest lessons we should have learned from our procedural programmer cousins is that trying to manage > 1 user of the same code leads to a excessively warm condition where the gatekeeper wears red pajamas and carries a pitchfork since every user needs to be updated simultaneously, or you end up with bunches of slightly different copies of code and no real idea who is using what.

The main thrust of the chapter was to demonstrate a number of interesting tools you can use to improve your databases for the convenience of the programmers and DBAs, as well as a number of ideas of how you might start to expand your own database implementations, including:

- *Numbers table*: A table of numbers that you can use to perform actions where a sequence of values might be useful. For example, finding gaps in another sequence of values (like identity values).
- *Calendar table*: A table that can help you use to implement methods of grouping date data using typical relational methods, both for normal date groupings like year, month, etc, but also for custom times like corporate fiscal calendars, sales, holiday, etc. While the most common usage might be for data warehouse/reporting systems, having the calendar table available in your OLTP database can be very useful/
- *Utilities*: Every programmer has code that they use to make their job easier. I presented conceptual utilities to monitor usage of the system and extended DDL to support operations that aren't part of the base DDL in T-SQL. Having them accessible in the database means that you can always be sure that the version that the programmer expected to be there will be.
- *Logging actions*: Utilities to log the actions of users in the database, generally for system management reasons. A common use is an error log to capture when and where errors are occurring.
- Any other use that you can dream up to make it easier for you, the programmers, and the DBAs to work with the database on any server, from development, to QA, and production. The more add-ins you can create that can make the experience across database implementations more consistent the better off you will be once your database is out in the world being used and, of course, maintained for years (likely decades) to come.

The goal is no and should always be to make the database its own universe. With the new contained database features in SQL Server 2012, and SQL Azure gaining traction, the database container is going to be closing up tighter than your grandmother's Tupperware casserole keeper. And even if it weren't, the more of the functionality that ends up in the database, the easier it will be to test that everything works, and that a change in one system will have no effect on the other. And that should mean fewer questions at 3:00 a.m. about why something failed because another database wasn't available when another needed it. That can't be a bad thing, can it?

www.it-ebooks.info

CHAPTER 13

Considering Data Access Strategies

Arguments are to be avoided; they are always vulgar and often convincing.

—Oscar Wilde

At this point in the process of covering the topic of database design, we have designed and implemented the database, devised effective security and indexing strategies, implemented concurrency controls, organized the database(s) into a manageable package and taken care of all of the other bits and pieces that go along with the task of creating a database. The next logical step is to decide on the data-access strategy and how best to implement and distribute data-centric business logic. Of course, in reality, design is not really a linear process, as performance tuning and indexing require a test plan, and data access is going to require that tools and some form of UI has been designed and that tools have been chosen to get an idea of how the data is going to be accessed. In this chapter I will take a rather brief look at some data access concerns and provide pros and cons of different methods…in other words opinions, technically-based opinions nevertheless.

Regardless of whether your application is a good, old-fashioned, client-server application, a multi-tier web application, uses an object relational mapper, or uses some new application pattern that hasn't been yet been created, data must be stored in and retrieved from tables. Therefore, most of the advice presented in this chapter will be relevant regardless of the type of application you're building. One way or another, you are going to have to build some interface between the data and the applications that use them. At minimum I will point out some of the pros and cons of the methods you will be using.

The really "good" arguments tend to get started about the topics in this chapter. Take every squabble to decide whether or not to use surrogate keys as primary keys, add to it all of the discussions about whether or not to use triggers in the application, and then multiply that by the number of managers it takes to screw in a light bulb, which I calculate as about 3.5. That is about how many times I have argued with a system implementer to decide whether stored procedures were a good idea.

Note The number of managers required to screw in a light bulb, depending on the size of the organization, would generally be a minimum of three or four: one manager to manage the person who notices the burnt out bulb, the manager of building services, and the shift manager of the person who actually changes light bulbs. Sometimes, the manager of the two managers might have to get involved to actually make things happen, so figure about three and a half.

---

www.it-ebooks.info

In this chapter, I will present a number of opinions on how to use stored procedures, ad hoc SQL, and the CLR. Each of these opinions is based on years of experience working with SQL Server technologies, but I am not so set in my ways that I cannot see the point of the people on the other side of any fence—anyone whose mind cannot be changed is no longer learning. So if you disagree with this chapter, feel free to e-mail me at louis@drsql.org; you won't hurt my feelings and we will both probably end up learning something in the process.

In this chapter, I am going to discuss the following topics:

- *Using ad hoc SQL*: Formulating queries in the application's presentation and manipulation layer (typically functional code stored in objects, such as .NET or Java, and run on a server or a client machine).
- *Using stored procedures*: Creating an interface between the presentation/manipulation layer and the data layer of the application. Note that views and functions, as well as procedures, also form part of this data-access interface. You can use all three of these object types.
- *Using CLR in T-SQL*: In this section, I will present some basic opinions on the usage of the CLR within the realm of T-SQL.

Each section will analyze some of the pros and cons of each approach, in terms of flexibility, security, performance, and so on. Along the way, I'll offer some personal opinions on optimal architecture and give advice on how best to implement both types of access.

■ **Note** You may also be thinking that another thing I might discuss is object-relational mapping tools, like Hibernate, Spring, or even the ADO.NET Entity Framework. In the end, however, these tools are really using ad hoc access, in that they are generating SQL on the fly. For the sake of this book, they should be lumped into the ad hoc group, unless they are used with stored procedures (which is pretty rare).

The most difficult part of a discussion of this sort is that the actual arguments that go on are not so much about right and wrong, but rather the question of which method is easier to program and maintain. SQL Server and Visual Studio .NET give you lots of handy-dandy tools to build your applications, mapping objects to data, and as the years pass, this becomes even truer with the Entity Framework and many of the object relational mapping tools out there.

The problem is that these tools don't always take enough advantage of SQL Server's best practices to build applications in their most common form of usage. Doing things in a best practice manner would mean doing a lot of coding manually, without the ease of automated tools to help you. Some organizations do this manual work with great results, but such work is rarely going to be popular with developers who have never hit the wall of having to support an application that is extremely hard to optimize once the system is in production.

A point that I really should make clear is that I feel that the choice of data-access strategy shouldn't be linked to the methods used for data validation nor should it be linked to whether you use (or how much you use) check constraints, triggers, and such. If you have read the entire book, you should be kind of tired of hearing how much I feel that you should do every possible data validation on the SQL Server data that can be done without making a maintenance nightmare. Fundamental data rules that are cast in stone should be done on the database server in constraints and triggers at all times so that these rules can be trusted by the user (for example, an ETL process). On the other hand, procedures or client code is going to be used to enforce a lot of the same rules, plus all of the mutable business rules too, but in either situation, non-data tier rules can be easily circumvented by using a different access path. Even database rules can be circumvented using bulk loading operations, so be careful there too.

www.it-ebooks.info

■ **Note** While I stand by all of the concepts and opinions in this entire book (typos not withstanding), I definitely do not suggest that your educational journey end here. Please read other people's work, try out everything, and form your own opinions. If some day you end up writing a competitive book to mine, the worst thing that happens is that people have another resource to turn to.

Ad Hoc SQL

Ad hoc SQL is sometimes referred to as "straight SQL" and generally refers to the formulation of `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements (as well as any others) in the client. These statements are then sent to SQL Server either individually or in batches of multiple statements to be syntax checked, compiled, optimized (producing a plan), and executed. SQL Server may use a cached plan from a previous execution, but it will have to pretty much exactly match the text of one call to another to do so, the only difference can be some parameterization of literals, which we will discuss a little later in the chapter.

I will make no distinction between ad hoc calls that are generated manually and those that use a middleware setup like LINQ: from SQL Server's standpoint, a string of characters is sent to the server and interpreted at runtime. So whether your method of generating these statements is good or poor is of no concern to me in *this* discussion, as long as the SQL generated is well formed and protected from users' malicious actions. (For example, injection attacks are generally the biggest offender. The reason I don't care where the ad hoc statements come from is that the advantages and disadvantages for the database support professionals are pretty much the same, and in fact, statements generated from a middleware tool can be worse, because you may not be able to change the format or makeup of the statements, leaving you with no easy way to tune statements, even if you can modify the source code.

Sending queries as strings of text is the way that most tools tend to converse with SQL Server, and is, for example, how SQL Server Management Studio does all of its interaction with the server metadata. If you have never used Profiler to watch the SQL that any of management and development tools uses, you should; just don't use it as your guide for building your OLTP system. It is, however, a good way to learn where some bits of metadata that you can't figure out come from.

There's no question that users will perform some ad hoc queries against your system, especially when you simply want to write a query and execute it just once. However, the more pertinent question is: should you be using ad hoc SQL when building the permanent interface to an OLTP system's data?

■ **Note** This topic doesn't include ad hoc SQL statements executed from stored procedures (commonly called dynamic SQL), which I'll discuss in the section "Stored Procedures."

Advantages

Using uncompiled ad hoc SQL has the following advantages over building compiled stored procedures:

- *Runtime control over queries*: Queries are built at runtime, without having to know every possible query that might be executed. This can lead to better performance as queries can be formed at runtime; you can retrieve only necessary data for `SELECT` queries or modify data that's changed for `UPDATE` operations.
- *Flexibility over shared plans and parameterization*: Because you have control over the queries, you can more easily build queries at runtime that use the same plans and even can be parameterized as desired, based on the situation.

www.it-ebooks.info

Runtime Control over Queries

Unlike stored procedures, which are prebuilt and stored in the SQL Server system tables, ad hoc SQL is formed at the time it's needed: at runtime. Hence, it doesn't suffer from some of the inflexible requirements of stored procedures. For example, say you want to build a user interface to a list of customers. You can add several columns to the SELECT clause, based on the tables listed in the FROM clause. It's simple to build a list of columns into the user interface that the user can use to customize his or her own list. Then the program can issue the list request with only the columns in the SELECT list that are requested by the user. Because some columns might be large and contain quite a bit of data, it's better to send back only the columns that the user really desires instead of also including a bunch of columns the user doesn't care about.

For instance, consider that you have the following table to document contacts to prospective customers (it's barebones for this example). In each query, you might return the primary key but show or not show it to the user based on whether the primary key is implemented as a surrogate or natural key—it isn't important to our example either way. You can create this table in any database you like. In the sample code, I've created a database named architectureChapter.

```
CREATE SCHEMA sales;
GO
CREATE TABLE sales.contact
(
  contactId int CONSTRAINT PKsales_contact PRIMARY KEY,
  firstName varchar(30),
  lastName varchar(30),
  companyName varchar(100),
  salesLevelId int, --real table would implement as a foreign key
  contactNotes varchar(max),
  CONSTRAINT AKsales_contact UNIQUE (firstName, lastName, companyName)
);
--a few rows to show some output from queries
INSERT INTO sales.contact
  (contactId, firstName, lastname, companyName, saleslevelId, contactNotes)
VALUES( 1,'Drue','Karry','SeeBeeEss',1,
           REPLICATE ('Blah…',10) + 'Called and discussed new ideas'),
      ( 2,'Jon','Rettre','Daughter Inc',2,
           REPLICATE ('Yada…',10) + 'Called, but he had passed on');
```

One user might want to see the person's name and the company, plus the end of the contactNotes, in his or her view of the data:

```
SELECT contactId, firstName, lastName, companyName,
       RIGHT(contactNotes,30) AS notesEnd
FROM sales.contact;
```

So something like:

```
contactId   firstName   lastName   companyName   notesEnd
---------   ---------   --------   -----------   ------------------------------
1           Drue        Karry      SeeBeeEss     Called and discussed new ideas
2           Jon         Rettre     Daughter Inc  ..Called, but he had passed on
```


www.it-ebooks.info

Another user might want (or need) to see less:

```
SELECT  contactId, firstName, lastName, companyName
FROM    sales.contact;
```

Which returns:

```
contactId       firstName       lastName        companyName
---------       --------        --------        -----------
1               Drue            Karry           SeeBeeEss
2               Jon             Rettre          Daughter Inc
```

And yet another may want to see all columns in the table, plus maybe some additional information. Allowing the user to choose the columns for output can be useful. Consider how the file-listing dialog works in Windows, as shown in Figure 13-1.

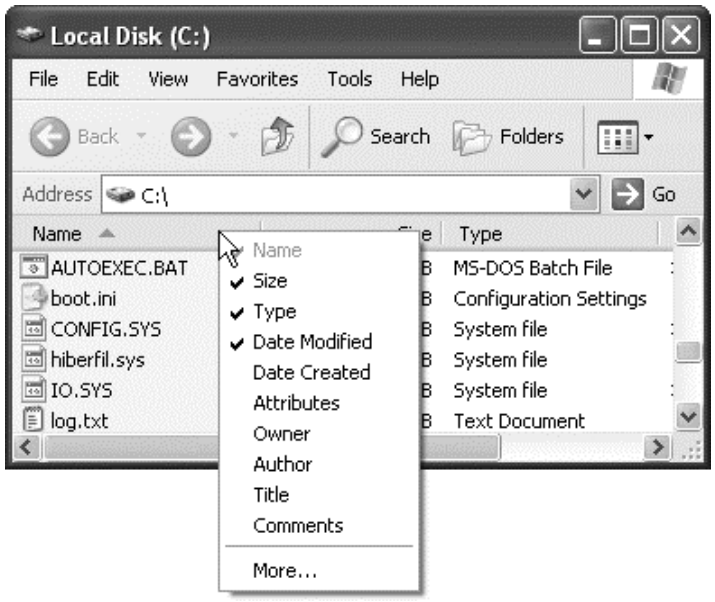


Figure 13-1. *The Windows file-listing dialog*

You can see as many or as few of the attributes of a file in the list as you like, based on some metadata you set on the directory. This is a useful method of letting the users choose what they want to see. Let's take this one step further. Consider that the `contact` table is then related to a table that tells us if a contact has purchased something:

```
CREATE TABLE sales.purchase
(
  purchaseId int CONSTRAINT PKsales_purchase PRIMARY KEY,
  amount numeric(10,2),
  purchaseDate date,
  contactId int
```


www.it-ebooks.info

```
    CONSTRAINT FKsales_contact$hasPurchasesIn$sales_purchase
    REFERENCES sales.contact(contactId)
);
INSERT INTO sales.purchase(purchaseId, amount, purchaseDate, contactId)
VALUES (1,100.00,'2012-05-12',1),(2,200.00,'2012-05-10',1),
       (3,100.00,'2012-05-12',2),(4,300.00,'2012-05-12',1),
       (5,100.00,'2012-04-11',1),(6,5500.00,'2012-05-14',2),
       (7,100.00,'2012-04-01',1),(8,1020.00,'2012-06-03',2);
```

Now consider that you want to calculate the sales totals and dates for the contact and add these columns to the allowed pool of choices. By tailoring the output when transmitting the results of the query back to the user, you can save bandwidth, CPU, and disk I/O. As I've stressed, values such as this should usually be calculated rather than stored, especially when working on an OLTP system.

In this case, consider the following two possibilities. If the user asks for a sales summary column, the client will send the whole query:

```
SELECT contact.contactId, contact.firstName, contact.lastName,
              sales.yearToDateSales, sales.lastSaleDate
FROM sales.contact AS contact
        LEFT OUTER JOIN
           (SELECT contactId,
                   SUM(amount) AS yearToDateSales,
                   MAX(purchaseDate) AS lastSaleDate
           FROM sales.purchase
           WHERE purchaseDate >= --the first day of the current year
                    DATEADD(day, 0, DATEDIFF(day, 0, SYSDATETIME() )
                      - DATEPART(DAYOFYEAR,SYSDATETIME() ) + 1)
           GROUP BY contactId) AS sales
           ON contact.contactId = sales.contactId
WHERE contact.lastName LIKE 'Rett%';
```

Which returns:

```
contactId        firstName         lastName         yearToDateSales    lastSaleDate
---------        ---------         --------         ---------------    ------------
2                Jon               Rettre           6620.00            2012-06-03
```

If the user doesn't ask for a sales summary column, the client will send only the bolded query:

```
SELECT contact.contactId, contact.firstName, contact.lastName
         -- ,sales.yearToDateSales, sales.lastSaleDate
FROM sales.contact AS contact
        --LEFT OUTER JOIN
        --   (SELECT contactId,
        --           SUM(amount) AS yearToDateSales,
        --           MAX(purchaseDate) AS lastSaleDate
        --    FROM   sales.purchase
        --    WHERE  purchaseDate >= --the first day of the current year
        --              DATEADD(DAY, 0, DATEDIFF(DAY, 0, SYSDATETIME() )
        --                - DATEPART(dayofyear,SYSDATETIME() ) + 1)
```


www.it-ebooks.info

```
        --    GROUP  by contactId) AS sales
        --    ON contact.contactId = sales.contactId
WHERE contact.lastName LIKE 'Karr%';
```

Which only returns:

```
contactId             firstName           lastName
----------            ---------           ---------
1                     Drue                Karry
```

Not wasting the resources to do calculations that aren't needed can save a lot of system resources if the aggregates in the derived table were very costly to execute

In the same vein, when using ad hoc calls, it's trivial (from a SQL standpoint) to build UPDATE statements that include only the columns that have changed in the set lists, rather than updating all columns, as can be necessary for a stored procedure. For example, take the customer columns from earlier: customerId, name, and number. You could just update all columns:

```
UPDATE sales.contact
SET    firstName = 'Drew',
       lastName = 'Carey',
       salesLevelId = 1, --no change
       companyName = 'CBS',
       contactNotes = 'Blah…Blah…Blah…Blah…Blah…Blah…Blah…Blah…Blah…'
                      + 'Blah…Called and discussed new ideas' --no change
WHERE contactId = 1;
```

But what if only the firstName and lastName columns change? What if the company column is part of an index, and it has data validations that take three seconds to execute? How do you deal with varchar(max) columns (or other long types)? Say the notes columns for contactId = 1 contain 3 MB each. Execution could take far more time than is desirable if the application passes the entire value back and forth each time. Using ad hoc SQL, to update the firstName column only, you can simply execute the following code:

```
UPDATE sales.contact
SET    firstName = 'John',
       lastName = 'Ritter'
WHERE  contactId = 1;
```

Some of this can be done with dynamic SQL calls built into the stored procedure, but it's far easier to know if data changed right at the source where the data is being edited, rather than having to check the data beforehand. For example, you could have every data-bound control implement a "data changed" property, and perform a column update only when the original value doesn't match the value currently displayed. In a stored-procedure-only architecture, having multiple update procedures is not necessarily out of the question, particularly when it is very costly to modify a given column.

One place where using ad hoc SQL can produce more reasonable code is in the area of optional parameters. Say that, in your query to the sales.contact table, your UI allowed you to filter on either firstName, lastName, or both. For example, take the following code to filter on both firstName and lastName:

```
SELECT  firstName, lastName, companyName
FROM    sales.contact
WHERE   firstName LIKE 'J%'
  AND lastName LIKE 'R%';
```


www.it-ebooks.info

What if the user only needed to filter by last name? Sending the '%' wildcard for firstName can cause code to perform less than adequately, especially when the query is parameterized. (I'll cover query parameterization in the next section, "Performance.")

```
SELECT  firstName, lastName, companyName
FROM    sales.contact
WHERE   firstName LIKE '%'
  AND lastName LIKE 'Carey%';
```

If you think this looks like a very silly query to execute, you are right. If you were writing this query, you would write the more logical version of this query, without the superfluous condition:

```
SELECT  firstName, lastName, companyName
FROM    sales.contact
WHERE   lastName LIKE 'Carey%';
```

This doesn't require any difficult coding. Just remove one of the criteria from the WHERE clause, and the optimizer needn't consider the other. What if you want to OR the criteria instead? Simply build the query with OR instead of AND. This kind of flexibility is one of the biggest positives to using ad hoc SQL calls.

■ **Note** The ability to change the statement programmatically does sort of play to the downside of any dynamically built statement, as now, with just two parameters, we have three possible variants of the statement to be used, so we have to consider performance for all three when we are building our test cases.

For a stored procedure, you might need to write code that functionally works in a manner such as the following:

```
IF @firstNameValue <> '%'
       SELECT firstName, lastName, companyName
       FROM   sales.contact
       WHERE  firstName LIKE @firstNameValue
         AND  lastName LIKE @lastNameValue;
ELSE
       SELECT firstName, lastName, companyName
       FROM   sales.contact
       WHERE  lastName LIKE @lastNameValue;
```

Or do something messy like this in your WHERE clause so if there is any value passed in it uses it, or uses '%' otherwise:

```
WHERE Firstname LIKE ISNULL(NULLIF(LTRIM(@FirstNamevalue) +'%','%'),Firstname)
  and Lastname LIKE ISNULL(NULLIF(LTRIM(@LastNamevalue) +'%','%'),Lastname)
```

Unfortunately though, this often does not optimize very well because the optimizer has a hard time optimizing for factors that can change based on different values of a variable—leading to the need for the branching solution mentioned previously to optimize for specific parameter cases. A better way to do this with stored procedures might be to create two stored procedures—one with the first query and another with the second query—especially if you need extremely high performance access to the data. You'd change this to the following code:

```
IF @firstNameValue <> '%'
       EXECUTE sales.contact$get @firstNameValue, @lastNameValue;
```


www.it-ebooks.info

```
ELSE
       EXECUTE sales.contact$getLastOnly @lastNameValue;
```

You can do some of this kind of ad hoc SQL writing using dynamic SQL in stored procedures. However, you might have to do a good bit of these sorts of IF blocks to arrive at which parameters aren't applicable in various datatypes. Because you know which parameters are applicable, due to knowing what the user filled in, it can be far easier to handle this situation using ad hoc SQL. Getting this kind of flexibility is the main reason that I use an ad hoc SQL call in an application (usually embedded in a stored procedure): you can omit parts of queries that don't make sense in some cases, and it's easier to avoid executing unnecessary code.

Flexibility over Shared Plans and Parameterization

Queries formed at runtime, using proper techniques, can actually be better for performance in many ways than using stored procedures. Because you have control over the queries, you can more easily build queries at runtime that use the same plans, and even can be parameterized as desired, based on the situation.

This is not to say that it is the most favorable way of implementing parameterization. (If you want to know the whole picture you have to read the whole section on ad hoc and stored procedures.) However, the fact is that ad hoc access tends to get a bad reputation for something that Microsoft fixed several versions back. In the following sections, Shared Execution Plans and Parameterization, I will take a look at the good points and the caveats you will deal with when building ad hoc queries and executing them on the server.

Shared Execution Plans

The age-old reason that people used stored procedures was because the query processor cached their plans. Every time you executed a procedure, you didn't have to decide the best way to execute the query. As of SQL Server 7.0 (which, was released in 1998!), cached plans were extended to include ad hoc SQL. However, the standard for what can be cached is pretty strict. For two calls to the server to use the same plan, the statements that are sent must be identical, except possibly for the literal values in search arguments. Identical means identical; add a comment, change the case, or even add a space character, and the plan will no longer match. SQL Server can build query plans that have parameters, which allow plan reuse by subsequent calls. However, overall, stored procedures are better when it comes to using cached plans for performance, primarily because the matching and parameterization are easier for the optimizer to do, since it can be done by `object_id`, rather than having to match larger blobs of text.

A fairly major caveat is that for ad hoc queries to use the same plan, they must be exactly the same, other than any values that can be parameterized. For example, consider the following two queries. (I'm using `AdventureWorks2012` tables for this example, as that database has a nice amount of data to work with.)

```
SELECT address.AddressLine1, address.AddressLine2,
        address.City, state.StateProvinceCode, address.PostalCode
FROM   Person.address as address
         JOIN Person.StateProvince as state
                on address.StateProvinceID = state.StateProvinceID
WHERE  address.addressLine1 = '1, rue Pierre-Demoulin';
```

Next, run the following query. See whether you can spot the difference between the two queries.

```
SELECT address.AddressLine1, address.AddressLine2,
        address.City, state.StateProvinceCode, address.PostalCode
FROM   Person.Address AS address
         JOIN Person.StateProvince AS state
                on address.StateProvinceID = state.StateProvinceID
WHERE  address.AddressLine1 = '1, rue Pierre-Demoulin';
```


www.it-ebooks.info

These queries can't share plans because the AS in the first query's FROM clause is lowercase (FROM Person.Address as address); but in the second, it's uppercase. Using the sys.dm_exec_query_stats dynamic management view, you can see that the case difference does cause two plans by running:

```
SELECT *
FROM   (SELECT execution_count,
           SUBSTRING(st.text, (qs.statement_start_offset/2)+1,
               ((CASE qs.statement_end_offset
                 WHEN -1 THEN DATALENGTH(st.text)
                ELSE qs.statement_end_offset
                END - qs.statement_start_offset)/2) + 1) AS statement_text
       FROM sys.dm_exec_query_stats AS qs
                CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
     ) AS queryStats
WHERE queryStats.statement_text LIKE 'SELECT address.AddressLine1%';
```

This SELECT statement will return at least two rows; one for each query you have just executed. (It could be more depending on whether or not you have executed the statement in this statement more than two times). Hence, trying to use some method to make sure that every query sent that is essentially the same query is formatted the same is important: queries must use the same format, capitalization, and so forth.

Parameterization

The next performance query plan topic to discuss is parameterization. When a query is parameterized, only one version of the plan is needed to service many queries. Stored procedures are parameterized in all cases, but SQL Server does parameterize ad hoc SQL statements. By default, the optimizer doesn't parameterize most queries, and caches most plans as straight text, unless the query is "simple." For example, it can only reference a single table (search for "Forced Parameterization" in Books Online for the complete details). When the query meets the strict requirements, it changes each literal it finds in the query string into a parameter. The next time the query is executed with different literal values, the same plan can be used. For example, take this simpler form of the previous query:

```
SELECT  address.AddressLine1, address.AddressLine2
FROM    Person.Address AS address
WHERE   address.AddressLine1 = '1, rue Pierre-Demoulin';
```

The plan (from using SHOWPLAN_TEXT ON in the manner we introduced in Chapter 10 "Basic Index Usage Patterns") is as follows:

```
|--Index Seek(OBJECT:([AdventureWorks2012].[Person].[Address].|
   IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode] AS [address]),
   SEEK:([address].[AddressLine1]=CONVERT_IMPLICIT(nvarchar(4000),[@1],0)) ORDERED FORWARD)
```

The value of N'1, rue Pierre-Demoulin' has been changed to @1 (which is in bold in the plan), and the value is filled in from the literal at execute time. However, try executing this query that accesses two tables:

```
SELECT address.AddressLine1, address.AddressLine2,
       address.City, state.StateProvinceCode, address.PostalCode
FROM   Person.Address AS address
         JOIN Person.StateProvince AS state
               ON address.StateProvinceID = state.StateProvinceID
WHERE  address.AddressLine1 ='1, rue Pierre-Demoulin';
```


www.it-ebooks.info

The plan won't recognize the literal and parameterize it:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([address].[StateProvinceID]))
      |--Index Seek(OBJECT:([AdventureWorks].[Person].[Address].
           [IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode]
                                                                    AS [address]),
SEEK:([address].[AddressLine1]=N'1, rue Pierre-Demoulin')
                                                        ORDERED FORWARD)
      |--Clustered Index Seek(OBJECT:([AdventureWorks2008].[Person].[StateProvince].
           [PK_StateProvince_StateProvinceID] AS [state]),
              SEEK:([state].[StateProvinceID]=[AdventureWorks].[Person].
                  [Address].[StateProvinceID] as [address].[StateProvinceID]) ORDERED FORWARD)
```

Note that the literal (bolded in this plan) from the query is still in the plan, rather than a parameter. Both plans are cached, but the first one can be used regardless of the literal value included in the `WHERE` clause. In the second, the plan won't be reused unless the precise literal value of `N'1, rue Pierre-Demoulin'` is passed in.

If you want the optimizer to be more liberal in parameterizing queries, you can use the `ALTER DATABASE` command to force the optimizer to parameterize:

```
ALTER DATABASE AdventureWorks2012
    SET PARAMETERIZATION FORCED;
```

Try the plan of the query with the join. It now has replaced the `N'1, rue Pierre-Demoulin'` with `CONVERT_IMPLICIT(nvarchar(4000),[@0],0)`. Now the query processor can reuse this plan no matter what the value for the literal is. This can be a costly operation in comparison to normal, text-only plans, so not every system should use this setting. However, if your system is running the same, reasonably complex-looking queries over and over, this can be a wonderful setting to avoid the need to pay for the query optimization.

```
 |--Nested Loops(Inner Join, OUTER REFERENCES:([address].[StateProvinceID]))
     |--Index Seek(OBJECT:([AdventureWorks2012].[Person].[Address].
        [IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode]
                                                                  AS [address]),
SEEK:([address].[AddressLine1]=CONVERT_IMPLICIT(nvarchar(4000),[@0],0))
                                                              ORDERED FORWARD)
      |--Clustered Index Seek(OBJECT:([AdventureWorks2012].[Person].[StateProvince].
   [PK_StateProvince_StateProvinceID] AS [state]),
      SEEK:([state].[StateProvinceID]=[AdventureWorks2012].[Person].
      [Address].[StateProvinceID] as [address].[StateProvinceID]
```

Not every query will be parameterized when forced parameterization is enabled. For example, change the equality to a `LIKE` condition:

```
SELECT address.AddressLine1, address.AddressLine2,
       address.City, state.StateProvinceCode, address.PostalCode
FROM   Person.Address AS address
         JOIN Person.StateProvince AS state
               ON address.StateProvinceID = state.StateProvinceID
WHERE  address.AddressLine1 LIKE '1, rue Pierre-Demoulin';
```

The plan will contains the literal, rather than the parameter, because it cannot parameterize the second and third arguments of the `LIKE` operator (the arguments are `arg1 LIKE arg2 [ESCAPE arg3]`).

www.it-ebooks.info

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([address].[StateProvinceID]))
     |--Index Seek(OBJECT:([AdventureWorks2012].[Person].[Address].
[IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode]
    AS [address]),
SEEK:([address].[AddressLine1] >= N'1, rue Pierre-Demoulin'
              AND [address].[AddressLine1] <= N'1, ru …(cut off in view)
|--Clustered Index Seek(OBJECT:([AdventureWorks2012].[Person].[StateProvince].
                      [PK_StateProvince_StateProvinceID] AS [state]),
    SEEK:([state].[StateProvinceID]=[AdventureWorks2012].[Person].
    [Address].[StateProvinceID] AS [address].[StateProvinceID]
```

If you change the query to end with `WHERE '1, rue Pierre-Demoulin' LIKE address.AddressLine1`, it would be parameterized, but that construct is rarely what is desired.

For your applications, another method is to use parameterized calls from the data access layer. Basically using ADO.NET, this would entail using T-SQL variables in your query strings, and then using a `SqlCommand` object and its `Parameters` collection. The plan that will be created from SQL parameterized on the client will in turn be parameterized in the plan that is saved.

The myth that performance is definitely worse with ad hoc calls is just not quite true (certainly after 7.0). Performance can actually be less of a worry than you might have been led to believe when using ad hoc calls to the SQL Server in your applications. However, don't stop reading here. While performance may not suffer tremendously, performance tuning is one of the pitfalls, since once you have compiled that query into your application, changing the query is never as easy as it might seem during the development cycle.

So far, we have just executed queries directly, but there is a better method when building your interfaces that allows you to parameterize queries in a very safe manner. Using `sp_executeSQL`, you can fashion your SQL statement using variables to parameterize the query:

```
DECLARE @AddressLine1 nvarchar(60) = '1, rue Pierre-Demoulin',
       @Query nvarchar(500),
        @Parameters nvarchar(500)

SET @Query= N'SELECT address.AddressLine1, address.AddressLine2,address.City,
                    state.StateProvinceCode, address.PostalCode
              FROM   Person.Address AS address
                       JOIN Person.StateProvince AS state
                        ON address.StateProvinceID = state.StateProvinceID
            WHERE address.AddressLine1 LIKE @AddressLine1';
SET @Parameters = N'@AddressLine1 nvarchar(60)';

EXECUTE sp_executesql @Query, @Parameters, @AddressLine1 = @AddressLine1;
```

Using `sp_executesql` is generally considered the safest way to parameterize queries because it does a good job of parameterizing the query and helps avoid issues like SQL injection attacks, which I will cover later in this chapter.

Finally, if you know you need to reuse the query multiple times, you can compile it and save the plan for reuse. This is generally useful if you are going to have to call the same object over and over, Instead of `sp_executesql`, use `sp_prepare` to prepare the plan; only this time you won't use the actual value:

```
DECLARE @Query nvarchar(500),
        @Parameters nvarchar(500),
        @Handle int

SET @Query= N'SELECT address.AddressLine1, address.AddressLine2,address.City,
```


www.it-ebooks.info

```
                state.StateProvinceCode, address.PostalCode
        FROM  Person.Address AS address
                JOIN Person.StateProvince AS state
                 ON address.StateProvinceID = state.StateProvinceID
    WHERE address.AddressLine1 LIKE @AddressLine1';
SET @Parameters = N'@AddressLine1 nvarchar(60)';

EXECUTE sp_prepare @Handle output, @Parameters, @Query;
SELECT @handle;
```

That batch will return a value that corresponds to the prepared plan, in my case it was 1. This value is you handle to the plan that you can use with `sp_execute` on the same connection only. All you need to execute the query is the parameter values and the `sp_execute` statement, and you can use and reuse the plan as needed:

```
DECLARE @AddressLine1 nvarchar(60) = '1, rue Pierre-Demoulin';
EXECUTE sp_execute 1, @AddressLine1;

SET @AddressLine1 = '6387 Scenic Avenue';
EXECUTE sp_execute 1, @AddressLine1;
```

You can unprepare the statement using `sp_unprepare` and the handle number. It is fairly rare that anyone will manually execute `sp_prepare` and `sp_execute`, but it is very frequently built into engines that are built to manage ad hoc access for you. It can be good for performance, but it is a pain for troubleshooting because you have to decode what the handle 1 actually represents. You can release the plan using `sp_unprepare` with a parameter of the handle.

What you end up with is pretty much the same as a procedure for performance, but it has to be done every time you run your app, and it is scoped to a connection, not shared on all connections. The better solution for parameterizing complex statements is a stored procedure. Generally, the only way this makes sense as a best practice is when you cannot use procedures, perhaps because of your tool choice or using a third-party application. Why? Well, the fact is with stored procedures, the query code is stored on the server and is a layer of encapsulation that reduces coupling; but more on that in the stored procedure section.

Pitfalls

I'm glad you didn't stop reading at the end of the previous section, because although I have covered the good points of using ad hoc SQL, there are the following significant pitfalls, as well:

- Low cohesion, high coupling
- Batches of statements
- Security issues
- SQL injection
- Performance-tuning difficulties

Low Cohesion, High Coupling

The number-one pitfall of using ad hoc SQL as your interface relates to what you learned back in Programming 101: strive for high cohesion, low coupling. Cohesion means that the different parts of the system work together to form a meaningful unit. This is a good thing, as you don't want to include lots of irrelevant code in the system, or be all over the place. On the other hand, coupling refers to how connected the different parts of a system are to one another. It's considered bad when a change in one part of a system breaks other parts of a system. (If you aren't too familiar with these terms, I suggest you go to www.wikipedia.org and search for these terms. You should build all the code you create with these concepts in mind.)

www.it-ebooks.info

When issuing T-SQL statements directly from the application, the structures in the database are tied directly to the client interface. This sounds perfectly normal and acceptable at the beginning of a project, but it means that any change in database structure might require a change in the user interface. This makes making small changes to the system just as costly as large ones, because a full testing cycle is required.

■ **Note** When I started this section, I told you that I wouldn't make any distinction between toolsets used. This is still true. Whether you use a horribly, manually coded system or the best object-relational mapping system, the fact that the application tier knows and is built specifically with knowledge of the base structure of the database is an example of the application and data tiers being highly coupled. Though stored procedures are similarly inflexible, they are stored with the data, allowing the disparate systems to be decoupled: the code on the database tier can be structurally dependent on the objects in the same tier without completely sacrificing your loose coupling.

For example, consider that you've created an employee table, and you're storing the employee's spouse's name, as shown in Figure 13-2.

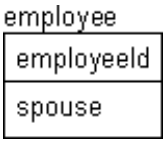


Figure 13-2. *An employee table*

Now, some new regulation requires that you have to include the ability to have more than one spouse, which necessitates a new table, as shown in Figure 13-3.

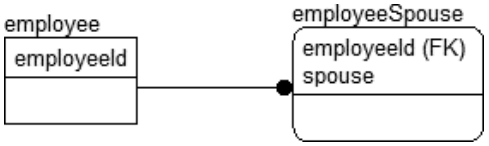


Figure 13-3. *Adding the ability to have more than one spouse*

The user interface must immediately be morphed to deal with this case, or at the least, you need to add some code to abstract this new way of storing data. In a scenario such as this, where the condition is quite rare (certainly most everyone will have zero or one spouse), a likely solution would be simply to encapsulate the one spouse into the `employee` table via a view, change the name of the object the non-data tier accesses, and the existing UI would still work. Then you can add support for the atypical case with some sort of `otherSpouse` functionality that would be used if the employee had more than one spouse. The original UI would continue to work, but a new form would be built for the case where `COUNT(spouse) > 1`.

Batches of More Than One Statement

A major problem with ad hoc SQL access is that when you need to do multiple commands and treat them as a single operation, it becomes increasingly more difficult to build the mechanisms in the application code to execute multiple statements as a batch, particularly when you need to group statements together in a transaction. When you have only

www.it-ebooks.info

individual statements, it's easy to manage ad hoc SQL for the most part. Some queries can get mighty complicated and difficult, but generally speaking, things pretty much work great when you have single statements per transaction. However, as complexity rises in the things you need to accomplish in a transaction, things get tougher. What about the case where you have 20 rows to insert, update, and/or delete at one time and all in one transaction?

Two different things usually occur in this case. The first way to deal with this situation is to start a transaction using functional code. For the most part, as discussed in Chapter 10, the best practice was stated as never to let transactions span batches, and you should minimize starting transactions using an ADO.NET object (or something like it). This isn't a hard and fast rule, as it's usually fine to do this with a middle-tier object that requires no user interaction. However, if something occurs during the execution of the object, you can still leave open connections and transactions to the server.

The second way to deal with this is to build a batching mechanism for batching SQL calls. Implementing the first method is self explanatory, but the second is to build a code-wrapping mechanism, such as the following:

```
BEGIN TRY
BEGIN TRANSACTION;

    <-- statements go here

COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    THROW 50000, '<describe what happened>',1;
END CATCH
```

For example, if you wanted to send a new invoice and line items, the application code would need to build a batch such as in the following code. Each of the SET @Action = … and INSERT statements would be put into the batch by the application with the rest being boilerplate code that is repeatable:

```
SET NOCOUNT ON;
BEGIN TRY
BEGIN TRANSACTION;
       DECLARE @Action nvarchar(200);

       SET @Action = 'Invoice Insert';
       INSERT invoice (columns) VALUES (values);

       SET @Action = 'First InvoiceLineItem Insert';
       INSERT invoiceLineItem (columns) VALUES (values);

       SET @Action = 'Second InvoiceLineItem Insert';
       INSERT invoiceLineItem (columns) VALUES (values);

       SET @Action = 'Third InvoiceLineItem Insert';
       INSERT invoiceLineItem (columns) VALUES (values);

COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    DECLARE @Msg nvarchar(4000);
    SET @Msg = @Action + ': Error was: ' + CAST(ERROR_NUMBER() as varchar(10)) + ':' +
                         ERROR_MESSAGE ();
    THROW 50000, @Msg,1 ;
END CATCH
```


www.it-ebooks.info

Executing multiple statements in a transaction is done on the server and the transaction either completes or not. There's no chance that you'll end up with a transaction swinging in the wind, ready to block the next user who needs to access the locked row (even in a table scan for unrelated items). A downside is that it does stop you from using any interim values from your statements (without some really tricky coding/forethought), but starting transactions outside of the batch you commit them is asking for blocking and locking due to longer batch times.

Starting the transaction outside of the server using application code is likely easier, but building this sort of batching interface is always the preferred way to go. First, it's better for concurrency, because only one batch needs to be executed instead of many little ones. Second, the execution of this batch won't have to wait on communications back and forth from the server before sending the next command. It's all there and happens in the single batch of statements.

■ **Note** The problem I have run into in almost all cases was that building a batch of multiple SQL statements is a very unnatural thing for the object oriented code to do. The way the code is generally set up is more one table to one object, where the `invoice` and `invoiceLineItem` objects have the responsibility of saving themselves. It is a lot easier to code, but too often issues come up in the execution of multiple statements and then the connection gets left open and other connections get blocked behind the locks that are left open due because of the open transaction.

Security Issues

Security is one of the biggest downsides to using ad hoc access. For a user to use his or her own Windows login to access the server, you have to grant too many rights to the system, whereas with stored procedures, you can simply give access to the stored procedures. Using ad hoc T-SQL, you have to go with one of three possible security patterns, each with their own downsides:

- *Use one login for the application*: This, of course, means that you have to code your own security system rather than using what SQL Server gives you. This even includes some form of login and password for application access, as well as individual object access.
- *Use application roles*: This is slightly better: while you have to implement security in your code since all application users will have the same database security, at the very least, you can let SQL server handle the data access via normal logins and passwords (probably using Windows authentication). Using application roles can be better way to give a user multiple sets of permissions as having only the one password for users to log in with usually avoids the mass of sticky notes embossed in bold letters all around the office: **payroll system ID: fred, password: fredsisawesome!**
- *Give the user direct access to the tables, or possibly views*: This unfortunately opens up your tables to users who discover the magical world of Management Studio, where they can open a table and immediately start editing it without any of those pesky UI data checks or business rules that only exist in your data access layer that I warned you about.

Usually, almost all applications follow the first of the three methods. Building your own security mechanisms is just considered a part of the process, and just about as often, it is considered part of the user interface's responsibilities. At the very least, by not giving all users direct access to the tables, the likelihood of them mucking around in the tables editing data all willy-nilly is greatly minimized. With procedures, you can give the users access to stored procedures, which are not natural for them to use and certainly would not allow them to accidentally delete data from a table.

The other issues security-wise are basically performance related. SQL Server must evaluate security for every object as it's used, rather than once at the object level for stored procedures—that is, if the owner of the

www.it-ebooks.info

procedure owns all objects. This isn't generally a big issue, but as your need for greater concurrency increases, everything becomes an issue!

■ **Caution** If you use the single-application login method, make sure not to use an account with system administration or database owner privileges. Doing so opens up your application to programmers making mistakes, and if you miss something that allows SQL injection attacks, which I describe in the next section, you could be in a world of hurt.

SQL Injection

A big issue with ad hoc query being hacked by a SQL injection attack. Unless you (and/or your toolset) program your ad hoc SQL intelligently and/or (mostly and) use the parameterizing methods we discussed earlier in this section on ad hoc SQL, a user could inject something such as the following:

```
' + CHAR(13) + CHAR(10) + ';SHUTDOWN WITH NOWAIT;' + '--'
```

In this case, the command might just shut down the server if the user has rights, but you can probably see far greater attack possibilities. I'll discuss more about injection attacks and how to avoid them in the "Stored Procedures" section. When using ad hoc SQL, you must be careful to avoid these types of issues for *every* call.

A SQL injection attack is not terribly hard to beat, but the fact is, for any use enterable text where you don't use some form of parameterization, you have to make sure to escape any single quote characters that a user passes in. For general text entry, like a name, commonly if the user passes in a string like "O'Malley", you know to change this to `'O''Malley'`. For example, consider the following batch, where the resulting query will fail. (I have to escape the single quote in the literal to allow the query to execute.)

```
DECLARE @value varchar(30) = 'O''Malley';
SELECT 'SELECT '''+ @value + '''';
EXECUTE ('SELECT '''+ @value + '''');
```

This will return:

```
SELECT 'O'Malley'

Msg 105, Level 15, State 1, Line 1
Unclosed quotation mark after the character string ''.
```

This is a very common problem that arises, and all too often the result is that the user learns not to enter a single quote in the query parameter value. But you can make sure this doesn't occur by changing all single quotes in the value to double single quotes, like in the `DECLARE @value` statement:

```
DECLARE @value varchar(30) = 'O''Malley', @query nvarchar(300);
SELECT @query = 'SELECT ' + QUOTENAME(@value,'''');
SELECT @query;
EXECUTE (@query );
```

This returns:

```
SELECT 'O''Malley'
------------------
O'Malley
```


www.it-ebooks.info

Now, if someone tries to put a single quote, semicolon, and some other statement in the value, it doesn't matter; it will always be treated as a literal string:

```
DECLARE @value varchar(30) = 'O''; SELECT ''badness',
        @query nvarchar(300);
SELECT  @query = 'SELECT ' + QUOTENAME(@value,'''');
SELECT  @query;
EXECUTE (@query );
```

The query is now, followed by the return:

```
SELECT 'O''; SELECT ''badness'
------------------------------
O'; SELECT 'badness
```

However, what isn't quite so obvious is that you have to do this for every single string, even if the string value could never legally (due to application and constraints) have a single quote in it. If you don't double up on the quotes, the person could put in a single quote and then a string of SQL commands—this is where you get hit by the injection attack. And as I said before, the safest method of avoiding injection issues is to always parameterize your queries; though it is very hard to use a variable query conditions using the parameterized method, losing some of the value of using ad hoc SQL.

```
DECLARE @value varchar(30) = 'O''; SELECT ''badness',
        @query nvarchar(300),
       @parameters nvarchar(200) = N'@value varchar(30)';
SELECT  @query = 'SELECT ' + QUOTENAME(@value,'''');
SELECT  @query;
EXECUTE sp_executesql @Query, @Parameters, @value = @value;
```

If you employ ad hoc SQL in your applications, I strongly suggest you do more reading on the subject of SQL injection, and then go in and look at the places where SQL commands can be sent to the server to make sure you are covered. SQL injection is especially dangerous if the accounts being used by your application have too much power because you didn't set up particularly granular security.

Note that if all of this talk of parameterization sounds complicated, it kind of is. Generally there are two ways that parameterization happens well. Either by building a framework that forces you to follow the correct pattern (as do most object-relational tools, thought they often force (or at least lead) you into sub-optimal patterns of execution, like dealing with every statement separately without transactions.) The second way is stored procedures. Stored procedures parameterize in a manner that is impervious to SQL injection (except, when you use dynamic SQL, which, as I will discuss later, is subject to the same issues as ad hoc access from any client.)

Difficulty Tuning for Performance

Performance tuning is generally far more difficult when having to deal with ad hoc requests, for a few reasons:

- *Unknown queries*: The application can be programmed to send any query it wants, in any way. Unless very extensive testing is done, slow or dangerous scenarios can slip through. With procedures, you have a tidy catalog of possible queries that might be executed. Of course, this concern can be mitigated by having a single module where SQL code can be created and a method to list all possible queries that the application layer can execute (however, that takes more discipline than most organizations have.).

www.it-ebooks.info

- *Often requires people from multiple teams*: It may seem silly, but when something is running slower, it is always blamed on SQL Server first. With ad hoc calls, the best thing that the database administrator can do is use a profiler to capture the query that is executed, see if an index could help, and call for a programmer, leading to the issue in the next bullet.
- *Recompile required for changing queries*: If you want to add a tip to a query to use an index, a rebuild and a redeploy are required. For stored procedures, you'd simply modify the query without the client knowing.

These reasons seem small during the development phase, but often they're the real killers for tuning, especially when you get a third-party application and its developers have implemented a dumb query that you could easily optimize, but since the code is hard coded in the application, modification of the query isn't possible (not that this regularly happens; no, not at all; nudge, nudge, wink, wink).

SQL Server 2005 gave us plan guides (and there are some improvements in their usability in later versions) that can be used to force a plan for queries, ad hoc calls, and procedures when you have troublesome queries that don't optimize naturally, but the fact is, going in and editing a query to tune is far easier than using plan guides.

Stored Procedures

Stored procedures are compiled batches of SQL code that can be parameterized to allow for easy reuse. The basic structure of stored procedures follows. (See SQL Server Books Online at `http://msdn.microsoft.com/en-us/library/ms130214.aspx` for a complete reference.)

```
CREATE PROCEDURE <procedureName>
[(
        @parameter1 <datatype> [ = <defaultvalue> [OUTPUT]]
        @parameter2 <datatype> [ = <defaultvalue> [OUTPUT]]
        …
        @parameterN <datatype> [ = <defaultvalue> [OUTPUT]]
)]
AS

<T-SQL statements> | <CLR Assembly reference>
```

There isn't much more to it. You can put any statements that could have been sent as ad hoc calls to the server into a stored procedure and call them as a reusable unit. You can return an integer value from the procedure by using the `RETURN` statement, or return almost any datatype by declaring the parameter as an `output` parameter other than `text` or `image`, though you should be using (`max`) datatypes instead because they will not likely exist in the version after 2012. (You can also not return a table valued parameter.) After the `AS`, you can execute any T-SQL commands you need to, using the parameters like variables.

The following is an example of a basic procedure to retrieve rows from a table (continuing to use the `AdventureWorks2012` tables for these examples):

```
CREATE PROCEDURE Person.Address$select
(
    @addressLine1 nvarchar(120) = '%',
    @city nvarchar(60) = '%',
    @state nchar(3) = '___', --special because it is a char column
    @postalCode nvarchar(8) = '%'
) AS
--simple procedure to execute a single query
SELECT address.AddressLine1, address.AddressLine2,
        address.City, state.StateProvinceCode, address.PostalCode
```


www.it-ebooks.info

```
FROM   Person.Address as address
         JOIN Person.StateProvince as state
                ON address.StateProvinceID = state.StateProvinceID
WHERE address.AddressLine1 LIKE @addressLine1
  AND address.City LIKE @city
  AND state.StateProvinceCode LIKE @state
  AND address.PostalCode LIKE @postalCode;
```

Now instead of having the client programs formulate a query by knowing the table structures, the client can simply issue a command, knowing a procedure name and the parameters. Clients can choose from four possible criteria to select the addresses they want. For example, they'd use the following if they want to find people in London:

```
Person.Address$select @city = 'London';
```

Or they could use the other parameters:

```
Person.Address$select @postalCode = '98%', @addressLine1 = '%Hilltop%';
```

The client doesn't know whether the database or the code is well built or even if it's horribly designed. Originally our state value might have been a part of the address table but changed to its own table when we realized that it was necessary to store more information about a state than a simple code. The same might be true for the `city`, the `postalCode`, and so on. Often, the tasks the client needs to do won't change based on the database structures, so why should the client need to know the database structures?

For much greater detail about how to write stored procedures and good T-SQL, consider the books *Pro T-SQL 2008 Programmer's Guide* by Michael Coles (Apress, 2008); *Inside Microsoft SQL Server 2008: T-SQL Programming (Pro-Developer)* by Itzik Ben-Gan, et al. (Microsoft Press, 2009); or search for "stored procedures" in Books Online. In this section, I'll look at some of the characteristics of using stored procedures as our only interface between client and data. I'll discuss the following topics:

- *Encapsulation*: Limits client knowledge of the database structure by providing a simple interface for known operations
- *Dynamic procedures*: Gives the best of both worlds, allowing for ad hoc–style code without giving ad hoc access to the database
- *Security*: Provides a well-formed interface for known operations that allows you to apply security only to this interface and disallow other access to the database
- *Performance*: Allows for efficient parameterization of any query, as well as tweaks to the performance of any query, without changes to the client interface
- *Pitfalls*: Drawbacks associated with the stored-procedure–access architecture

Encapsulation

To me, encapsulation is the primary reason for using stored procedures, and it's the leading reason behind all the other topics that I'll discuss. When talking about encapsulation, the idea is to hide the working details from processes that have no need to know about the details. Encapsulation is a large part of the desired "low coupling" of our code that I discussed in the pitfalls of ad hoc access. Some software is going to have to be coupled to the data structures, of course, but locating that code with the structures makes it easier to manage. Plus, the people who generally manage the SQL Server code on the server are not the same people who manage the compiled code.

For example, when we coded the `person.address$select` procedure, it's unimportant to the client how the procedure was coded. We could have built it based on a view and selected from it, or the procedure could call 16

www.it-ebooks.info

different stored procedures to improve performance for different parameter combinations. We could even have used the dreaded cursor version that you find in some production systems:

```
--pseudocode:
CREATE PROCEDURE person.address$select
…
Create temp table;
Declare cursor for (select all rows from the address table);
Fetch first row;
While not end of cursor (@@fetch_status)
  Begin
         Check columns for a match to parameters;
         If match, put into temp table;
         Fetch next row;
  End;

SELECT * FROM temp table;
```

This would be horrible, horrible code to be sure. I didn't give real code so it wouldn't be confused for a positive example and imitated. (Somebody would end up blaming me, though definitely not you!) However, it certainly could be built to return correct data and possibly could even be fast enough for smaller data sets. Even better, when the client executes the following code, they get the same result, regardless of the internal code:

```
Person.Address$select @city = 'london';
```

What makes procedures great is that you can rewrite the guts of the procedure using the server's native language without any concern for breaking any client code. This means that anything can change, including table structures, column names, and coding method (cursor, join, and so on), and no client code need change as long as the inputs and outputs stay the same.

The only caveat to this is that you can get some metadata about procedures only when they are written using compiled SQL without conditionals (if condition select … else select…). For example, using `sp_describe_first_result_set` you can metadata about what the procedure we wrote earlier returns:

```
EXECUTE sp_describe_first_result_set
           N'Person.Address$select @postalCode = ''98%'', @addressLine1 = ''%Hilltop%'';'
```

This returns metadata about what will be returned (this is just a small amount of what is returned):

```
column_ordinal        name                 system_type_name
--------------        -----------------    ----------------
 1                    AddressLine1         nvarchar(60)
 2                    AddressLine2         nvarchar(60)
 3                    City                 nvarchar(30)
 4                    StateProvinceCode    nchar(3)
 5                    PostalCode           nvarchar(15)
```

For poorly formed procedures, (even if it returns what appears to be the exact same result set, you will not be as able to get the metadata from the procedure. For example, consider the following procedure:

```
CREATE PROCEDURE test (@value int = 1)
AS
IF @value = 1
    SELECT 'FRED' AS name;
```


www.it-ebooks.info

```
ELSE
    SELECT 200 AS name;
```

If you run the procedure with 1 for the parameter, it will return FRED, for any other value, it will return 200. Both are named "name", but they are not the same type. So checking the result set:

```
EXECUTE sp_describe_first_result_set N'test'
```

Returns this (actually quite) excellent error message:

```
Msg 11512, Level 16, State 1, Procedure sp_describe_first_result_set, Line 1

The metadata could not be determined because the statement 'SELECT 'FRED' as name;' in
procedure 'test' is not compatible with the statement 'SELECT 200 as name;' in procedure
'test'.
```

This concept of having easy access to the code may seem like an insignificant consideration, especially if you generally only work with limited sized sets of data. The problem is, as data set sizes fluctuate, the types of queries that will work often vary greatly. When you start dealing with increasing orders of magnitude in the number of rows in your tables, queries that seemed just fine somewhere at ten thousand rows start to fail to produce the kinds of performance that you need, so you have to tweak the queries to get results in an amount of time that users won't complain to your boss about. I will cover more about performance tuning in a later section.

■ **Note** Some of the benefits of building your objects in the way that I describe can also be achieved by building a solid middle-tier architecture with a data layer that is flexible enough to deal with change. However, I will always argue that it is easier to build your data access layer in the T-SQL code that is built specifically for data access. Unfortunately, it doesn't solve the code ownership issues (functional versus relational programmers) nor does it solve the issue with performance-tuning the code.

Dynamic Procedures

You can dynamically create and execute code in a stored procedure, just like you can from the front end. Often, this is necessary when it's just too hard to get a good query using the rigid requirements of precompiled stored procedures. For example, say you need a procedure that needs a lot of optional parameters. It can be easier to include only parameters where the user passes in a value and let the compilation be done at execution time, especially if the procedure isn't used all that often. The same parameter sets will get their own plan saved in the plan cache anyhow, just like for typical ad hoc SQL.

Clearly, some of the problems of straight ad hoc SQL pertain here as well, most notably SQL injection. You must always make sure that no input users can enter can allow them to return their own results, allowing them to poke around your system without anyone knowing. As mentioned before, a common way to avoid this sort of thing is always to check the parameter values and immediately double up the single quotes so that the caller can't inject malicious code where it shouldn't be.

Make sure that any parameters that don't need quotes (such as numbers) are placed into the correct datatype. If you use a string value for a number, you can insert things such as 'novalue' and check for it in your code, but another user could put in the injection attack value and be in like Flynn. For example, take the sample

www.it-ebooks.info

procedure from earlier, and let's turn it into the most obvious version of a dynamic SQL statement in a stored procedure:

```
ALTER PROCEDURE Person.Address$select
(
    @addressLine1 nvarchar(120) = '%',
    @city         nvarchar(60) = '%',
    @state        nchar(3) = '___',
    @postalCode   nvarchar(50) = '%'
) AS
BEGIN
    DECLARE @query varchar(max);
    SET @query =
                'SELECT address.AddressLine1, address.AddressLine2,
                       address.City, state.StateProvinceCode, address.PostalCode
                 FROM Person.Address AS address
                       join Person.StateProvince AS state
                            on address.StateProvinceID = state.StateProvinceID
                 WHERE address.City LIKE ''' + @city + '''
                    AND state.StateProvinceCode LIKE ''' + @state + '''
                    AND address.PostalCode LIKE ''' + @postalCode + '''
                    --this param is last because it is largest
                    --to make the example
                    --easier as this column is very large
                    AND address.AddressLine1 LIKE ''' + @addressLine1 + '''';

    SELECT @query; --just for testing purposes
    EXECUTE (@query);
END;
```

There are two problems with this version of the procedure. The first is that you don't get the full benefit, because in the final query you can end up with useless parameters used as search arguments that make using indexes more difficult, which is one of the main reasons I use dynamic procedures. I'll fix that in the next version of the procedure, but the most important problem is the injection attack. For example, let's assume that the user who's running the application has `dbo` powers or rights to `sysusers`. The user executes the following statement:

```
EXECUTE Person.Address$select
                          @addressLine1 = '~''select name from sysusers--';
```

This returns three result sets: the two (including the test `SELECT`) from before plus a list of all of the users in the `AdventureWorks2012` database. No rows will be returned to the proper result sets, because no address lines happen to be equal to `'~'`, but the list of users is not a good thing because with some work, a decent hacker could probably figure out how to use a `UNION` and get back the users as part of the normal result set.

The easy way to correct this is to use the `quotename()` function to make sure that all values that need to be surrounded by single quotes are formatted in such a way that no matter what a user sends to the parameter, it cannot cause a problem. Note that if you programmatically chose columns that you ought to use the `quotename()` function to insert the bracket around the name. `SELECT quotename('FRED')` would return `[FRED]`.

In the next code block, I will change the procedure to safely deal with invalid quote characters, plus instead of just blindly using the parameters, if the parameter value is the same as the default, I will leave off the values

www.it-ebooks.info

from the WHERE clause. Note that using sp_executeSQL and parameterizing will not be possible if you want to do a variable WHERE or JOIN clause, so you have to take care to avoid SQL injection in the query itself.

```
ALTER PROCEDURE Person.Address$select
(
    @addressLine1 nvarchar(120) = '%',
    @city nvarchar(60) = '%',
    @state nchar(3) = '___',
    @postalCode nvarchar(50) = '%'
) AS
BEGIN
    DECLARE @query varchar(max);
    SET @query =
                'SELECT address.AddressLine1, address.AddressLine2,
                        address.City, state.StateProvinceCode, address.PostalCode
                 FROM Person.Address AS address
                        JOIN Person.StateProvince AS state
                                ON address.StateProvinceID = state.StateProvinceID
                 WHERE 1=1';
    IF @city <> '%'
          SET @query = @query + ' AND address.City LIKE ' + QUOTENAME(@city,'''');
    IF @state <> '___'
            SET @query = @query + ' AND state.StateProvinceCode LIKE ' + QUOTENAME(@state,'''');
    IF @postalCode <> '%'
            SET @query = @query + ' AND address.City LIKE ' + QUOTENAME(@city,'''');
    IF @addressLine1 <> '%'
            SET @query = @query + ' AND address.AddressLine1 LIKE ' +
                                          QUOTENAME(@addressLine1,'''');
    SELECT @query;
    EXECUTE (@query);
END;
```

Now you might get a much better plan, especially if there are several useful indexes on the table. That's because SQL Server can make the determination of what indexes to use at runtime based on the parameters needed, rather than using a single stored plan for every possible combination of parameters. You also don't have to worry about injection attacks, because it's impossible to put something into any parameter that will be anything other than a search argument, and that will execute any code other than what you expect. Basically this version of a stored procedure is the answer to the flexibility of using ad hoc SQL, though it is a bit messier to write. However, it is located right on the server where it can be tweaked as necessary.

Try executing the evil version of the query, and look at the WHERE clause it fashions:

```
WHERE   1=1
      AND address.AddressLine1 LIKE '~''select name from sysusers--'
```

The query that is formed, when executed will now just return two result sets (one for the query and one for the results), and no rows for the executed query. This is because you are looking for rows where address.addressLine1 is like ~'select name from sysusers--. While not being exactly impossible, this is certainly very, very unlikely.

I should also note that in versions of SQL Server before 2005, using dynamic SQL procedures would break the security chain, and you'd have to grant a lot of extra rights to objects just used in a stored procedure. This little fact was enough to make using dynamic SQL not a best practice for SQL Server 2000 and earlier versions. However, in SQL Server 2005 you no longer had to grant these extra rights, as I'll explain in the next section. (Hint: you can EXECUTE AS someone else.)

www.it-ebooks.info

Security

My second most favorite reason for using stored-procedure access is security. You can grant access to just the stored procedure, instead of giving users the rights to all the different resources used by the stored procedure. Granting rights to all the objects in the database gives them the ability to open Management Studio and do the same things (and more) that the application allows them to do. This is rarely the desired effect, as an untrained user let loose on base tables can wreak havoc on the data. ("Oooh, I should change that. Oooh, I should delete that row. Hey, weren't there more rows in this table before?") I should note that if the database is properly designed, users can't violate core structural business rules, but they can circumvent business rules in the middle tier and can execute poorly formed queries that chew up important resources.

■ **Note** In general, it is best to keep your users away from the power tools like Management Studio and keep them in a sandbox where even if they have advanced powers (like because they are CEO) they cannot accidentally see too much (and particularly modify and/or delete) data that they shouldn't. Provide tools that hold user's hands and keep them from shooting off their big toe (or really any toe for that matter).

With stored procedures, you have a far clearer surface area of a set of stored procedures on which to manage security at pretty much any granularity desired, rather than tables, columns, groups of rows (row-level security), and actions (`SELECT`, `UPDATE`, `INSERT`, `DELETE`), so you can give rights to just a single operation, in a single way. For example, the question of whether users should be able to delete a contact is wide open, but should they be able to delete their own contacts? Sure, so give them rights to execute `deletePersonalContact` (meaning a contact that the user owned). Making this choice easy would be based on how well you name your procedures. I use a naming convention of `<tablename | subject area>$<action>`. For example, to delete a contact, the procedure might be `contact$delete`, if users were allowed to delete any contact. How you name objects is completely a personal choice, as long as you follow a standard that is meaningful to you and others.

As discussed back in the ad hoc section, a lot of architects simply avoid this issue altogether by letting objects connect to the database as a single user (or being forced into this by political pressure, either way), and let the application handle security. That can be an adequate method of implementing security, and the security implications of this are the same for stored procedures or ad hoc usage. Using stored procedures still clarifies what you can or cannot apply security to.

In SQL Server 2005, the `EXECUTE AS` clause was added on the procedure declaration. In versions before SQL Server 2005, if a different user owned any object in the procedure (or function, view, or trigger), the caller of the procedure had to have explicit rights to the resource. This was particularly annoying when having to do some small dynamic SQL operation in a procedure, as discussed in the previous section.

The `EXECUTE AS` clause gives the programmer of the procedure the ability to build procedures where the procedure caller has the same rights in the procedure code as the owner of the procedure—or if permissions have been granted, the same rights as any user or login in the system.

For example, consider that you need to do a dynamic SQL call to a table. (In reality, it ought to be more complex, like needing to use a sensitive resource.) First, create a test user:

```
CREATE USER fred WITHOUT LOGIN;
```

Next, create a simple stored procedure:

```
CREATE PROCEDURE dbo.testChaining
AS
EXECUTE ('SELECT CustomerID, StoreID, AccountNumber
          FROM Sales.Customer');
```


www.it-ebooks.info

```
GO
GRANT EXECUTE ON testChaining TO fred;
```

Now execute the procedure (changing your security context to be this user):

```
EXECUTE AS user = 'fred';
EXECUTE dbo.testChaining;
REVERT;
```

You're greeted with the following error:

```
Msg 229, Level 14, State 5, Line 1
SELECT permission denied on object 'Customer', database 'AdventureWorks', schema 'Sales'.
```

You could grant rights to the user directly to the object, but this gives users more usage than just from this procedure, which is probably not desirable. Now change the procedure to `EXECUTE AS SELF`:

```
ALTER PROCEDURE dbo.testChaining
WITH EXECUTE AS SELF
AS
EXECUTE ('SELECT CustomerID, StoreId, AccountNumber
          FROM Sales.Customer');
```

■ **Note** If you have restored this database from the web, you may get an error message: Msg 15517, Cannot execute as the database principal because the principal "dbo" does not exist, this type of principal cannot be impersonated, or you do not have permission. This will occur if your database owner's `sid` is not correct for the instance you are working on. Use: `ALTER AUTHORIZATION ON DATABASE::AdventureWorks2012 to SA` to set the owner to the SA account. Determine database owner using query: `SELECT SUSER_SNAME(owner_sid) FROM sys.databases WHERE name = 'Adventureworks2012';`

Now, go back to the context of user `Fred` and try again. Just like when `Fred` had access directly, you get back data. You use `SELF` to set the context the same as the principal creating the procedure. `OWNER` is usually the same as `SELF`, and you can only specify a single user in the database (it can't be a group).

Warning: the `EXECUTE AS` clause can be abused if you are not extremely careful. Consider the following query, which is obviously a gross exaggeration of what you might hope to see but not beyond possibility:

```
CREATE PROCEDURE dbo.doAnything
(
    @query nvarchar(4000)
)
WITH EXECUTE AS SELF
AS
EXECUTE (@query);
```


www.it-ebooks.info

This procedure gives the person that has access to execute it full access to the database. Bear in mind that *any* query can be executed (`DROP TABLE`? Sure, why not?), easily allowing improper code to be executed on the database. Now, consider a little math problem; add the following items:

- `EXECUTE AS SELF`
- Client executing code as the database owner (a very bad, yet very typical practice)
- The code for `dbo.doAnything`
- An injection-susceptible procedure, with a parameter that can hold approximately 120 characters (the length of the `dbo.doAnything` procedure plus punctuation to create it)

What do you get? If you guessed no danger at all, please e-mail me your Social Security number, address, and a major credit card number. If you realize that the only one that you really have control over is the fourth one and that hackers, once the `dbo.doAnything` procedure was created, could execute any code they wanted as the owner of the database, you get the gold star. So be careful to block code open to injection.

■ **Tip** I am not suggesting that you should avoid the `EXECUTE AS` setting completely, just that its use must be scrutinized a bit more than the average stored procedure along the lines of when a `#temp` table is used. Why was `EXECUTE AS` used? Is the use proper? You must be careful to understand that in the wrong hands this command can be harmful to security.

Performance

There are a couple of reasons why stored procedures are great for performance:

- Parameterization of complex plans is controlled by you at design time rather than controlled by the optimizer at compile time.
- You can performance-tune your queries without making invasive program changes.

Parameterization of Complex Plans

Stored procedures, unlike ad hoc SQL, always have parameterized plans for maximum reuse of the plans. This lets you avoid the cost of recompilation, as well as the advanced costs of looking for parameters in the code. Any literals are always literal, and any variable is always a parameter. This can lead to some performance issues as well, as occasionally the plan for a stored procedure that gets picked by the optimizer might not be as good of a plan as might be picked for an ad hoc procedure.

The interesting thing here is that, although you can save the plan of a single query with ad hoc calls, with procedures you can save the plan for a large number of statements. With all the join types, possible tables, indexes, view text expansions, and so on, optimizing a query is a nontrivial task that might take quite a few milliseconds. Now, admittedly, when building a single-user application, you might say, "Who cares?" However, as user counts go up, the amount of time begins to add up. With stored procedures, this has to be done only once. (Or perhaps a bit more frequently; SQL Server can create multiple copies of the plan if the procedure is heavily used.)

Stored procedure parameterization uses the variables that you pass the first time the procedure is called to create the best plan. This process is known as *parameter sniffing*. This is fine and dandy, except when you have a situation where you have some values that will work nicely for a query but others that work pitifully. Two different values that are being searched for can end up creating two different plans. Often, this is where you might pass in a value that tells the query that there are no values, and SQL Server uses that value to build the plan. When you

www.it-ebooks.info

pass in a real value, it takes far too long to execute. Using `WITH RECOMPILE` at the object level or the `RECOMPILE` statement-level hint can avoid the problems of parameter sniffing, but then you have to wait for the plan to be created for each execute, which can be costly. It's possible to branch the code out to allow for both cases, but this can get costly if you have a couple of different scenarios to deal with. In still other cases, you can use an `OPTIMIZE FOR` to optimize for the common case when there are parameter values that produce less than adequate results, although presumably results that you can live with, not a plan that takes an hour or more to execute what normally takes milliseconds.

In some cases, the plan gets stale because of changes in the data, or even changes in the structure of the table. In stored procedures, SQL Server can recompile only the single statement in the plan that needs recompiled. While managing compilation and re-compilation can seem a bit complicated, there are a few caveats but they are very few and far between. You have several ways to manage the parameterization and you have direct access to the code to change it. For the person who has to deal with parameter issues, or really any sort of tuning issues, our next topic is about tuning procedures without changing the procedure's public interface. You can use the tricks mentioned to fix the performance issue without the client's knowledge.

Fine-Tuning Without Program Changes

Even if you didn't have the performance capabilities of parameterization for stored procedures (say every query in your procedure was forced to do dynamic SQL), the ability to fine-tune the queries in the stored procedure without making any changes to the client code is of incredible value. Of course, this is the value of encapsulation, but again, fine-tuning is such an important thing.

Often, a third-party system is purchased that doesn't use stored procedures. If you're a support person for this type of application, you know that there's little you can do other than to add an index here and there.

"But," you're probably thinking, "shouldn't the third party have planned for all possible cases?" Sure they should, because while the performance characteristics of a system with 10 rows might turn out to be identical to one with 10,000, there is the outlier, like the organization that pumped 10 million new rows per day into that system that was only expected to do 10,000 per day. The fact is, SQL Server is built to operate on a large variety of hardware in a large variety of conditions. A system running on a one-processor laptop with its slow disk subsystem behaves exactly like a RAID 10 system with 20 high-speed, 32 GB, solid-state drives, right? (Another gold star if you just said something witty about how dumb that sounded.)

The answer is no. In general, the performance characteristics of database systems vary wildly based on usage characteristics, hardware, and data sizing issues. By using stored procedures, it's possible to tweak how queries are written, as the needs change from small dataset to massive dataset. For example, I've seen many not so perfect queries that ran great with 10,000 rows, but when the needs grew to millions of rows, the queries ran for hours. Rewriting the queries using proper query techniques, or sometimes using temporary tables gave performance that was several orders of magnitude better. And I have had the converse be true, where I have removed temporary tables and consolidated queries into a single statement to get better performance. The user of the procedures did not even know.

This ability to fine-tune without program changes is very important. Particularly as a corporate developer, when the system is running slow and you identify the query or procedure causing the issue, fixing it can be either a one-hour task or a one-week task. If the problem is a procedure, you can modify, test, and distribute the one piece of code. Even following all of the rules of proper code management, your modify/test/distribute cycle can be a very fast operation. However, if application code has to be modified, you have to coordinate multiple groups (DBAs and programmers, at least) and discuss the problem and the code has to be rewritten and tested (for many more permutations of parameters and settings than for the one procedure).

For smaller organizations, it can be overly expensive to get a really good test area, so if the code doesn't work quite right in production, you can tune it easily then. Tuning a procedure is easy, even modification procedures, you can just execute the code in a transaction and roll it back. Keep changing the code until satisfied, compile the code in the production database, and move on to the next problem. (This is not best practice, but it is something I have to do from time to time.)

www.it-ebooks.info

Pitfalls

So far, everything has been all sunshine and lollipops for using stored procedures, but this isn't always the case. We need to consider the following pitfalls:

- The high initial effort to create procedures can be prohibitive.
- It isn't always easy to implement optional parameters in searches in an optimum manner.
- It's more difficult to affect only certain columns in an operation.

Another pitfall, which I won't cover in detail here, is cross-platform coding. If you're going to build a data layer that needs to be portable to different platforms such as Oracle or MySQL, this need for cross-platform coding can complicate your effort, although it can still be worthwhile in some cases.

High Initial Effort

Of all the pros and cons, initial effort is most often the straw that breaks the camel's proverbial back in the argument for or against stored procedures. Pretty much every time I've failed to get stored procedure access established as the method of access, this is the reason given. There are many tools out there that can map a database to objects or screens to reduce development time. The problem is that they suffer from some or all of the issues discussed in the ad hoc SQL pitfalls.

It's an indefensible stance that writing lots of stored procedures takes less time up front—quite often, it takes quite a bit more time for initial development. Writing stored procedures is definitely an extra step in the process of getting an application up and running.

An extra step takes extra time, and extra time means extra money. You see where this is going, because people like activities where they see results, not infrastructure. When a charismatic programmer comes in and promises results, it can be hard to back up claims that stored procedures are certainly the best way to go. The best defenses are knowing the pros and cons and, especially, understanding the application development infrastructure you'll be dealing with.

Difficulty Supporting Optional Parameters in Searches

I already mentioned something similar to optional parameters earlier when talking about dynamic SQL. In those examples, all of the parameters used simple LIKE parameters with character strings. But what about integer values? Or numeric ones? As mentioned earlier in the ad hoc sections, a possible solution is to pass NULL into the variable values by doing something along the lines of the following code:

```
WHERE (integerColumn = @integerColumn OR @integerColumn is NULL)
  AND (numericColumn = @numericColumn OR @numericColumn is NULL)
  AND (characterColumn LIKE @characterColumn);
```

Generally speaking, it's possible to come up with some scheme along these lines to implement optional parameters alongside the rigid needs of procedures in stored procedures. Note too that using `NULL` as your get everything parameter value means it is then hard to get only `NULL` values. For character strings you can use `LIKE '%'`. You can even use additional parameters to state: `@returnAllTypeFlag` to return all rows of a certain type.

It isn't possible to come up with a scheme, especially a scheme that can be optimized. However, you can always fall back on using dynamic SQL for these types of queries using optional parameters, just like I did in the ad hoc section. One thing that can help this process is to add the `WITH RECOMPILE` clause to the stored-procedure declaration. This tells the procedure to create a new plan for every execution of the procedure.

Although I try to avoid dynamic SQL because of the coding complexity and maintenance difficulties, if the set of columns you need to deal with is large, dynamic SQL can be the best way to deal with the situation. Using

www.it-ebooks.info

dynamically built stored procedures is generally the same speed as using ad hoc access from the client, so the benefits from encapsulation still exist.

Difficulty Affecting Only Certain Columns in an Operation

When you're coding stored procedures without dynamic SQL, the code you'll write is going to be pretty rigid. If you want to write a stored procedure to modify a row in the table created earlier in the chapter—`sales.contact`—you'd write something along the lines of this skeleton procedure (back in the `architectureChapter` database):

```
CREATE PROCEDURE sales.contact$update
(
    @contactId   int,
    @firstName    varchar(30),
    @lastName     varchar(30),
    @companyName varchar(100),
    @salesLevelId  int,
    @personalNotes varchar(max),
    @contactNotes  varchar(max)
)
AS
    DECLARE @entryTrancount int = @@trancount;

    BEGIN TRY
          UPDATE sales.contact
          SET    firstName = @firstName,
                 lastName = @lastName,
                 companyName = @companyName,
                 salesLevelId = @salesLevelId,
                 personalNotes = @personalNotes,
                 contactNotes = @contactNotes
          WHERE  contactId = @contactId;
    END TRY
    BEGIN CATCH
      IF @@trancount > 0
           ROLLBACK TRANSACTION;

      DECLARE @ERRORmessage nvarchar(4000)
      SET @ERRORmessage = 'Error occurred in procedure ''' +
                  object_name(@@procid) + ''', Original Message: '''
                  + ERROR_MESSAGE() + '''';
      THROW 50000,@ERRORmessage,1;
END CATCH
```

A procedure such as this is fine most of the time, because it usually isn't a big performance concern just to pass all values and modify those values, even setting them to the same value and revalidating. However, in some cases, validating every column can be a performance issue because not every validation is the same as the next.

For example, say that the `salesLevelId` column was a very important column for the corporate sales process. And it needed to validate in the sales data if the customer could, in fact, actually be that level. A trigger might be created to do that validation, and it could take a relatively large amount of time. Note that when the average operation takes 1 millisecond, 100 milliseconds can actually be "a long time." It is all relative to what else

www.it-ebooks.info

is taking place and how many times a minute things are occurring. You could easily turn this into a dynamic SQL procedure, though since you don't know if the value of `salesLevel` has changed, you will have to check that first:

```
ALTER PROCEDURE sales.contact$update
(
    @contactId   int,
    @firstName    varchar(30),
    @lastName     varchar(30),
    @companyName varchar(100),
    @salesLevelId  int,
    @personalNotes varchar(max),
    @contactNotes  varchar(max)
)
WITH EXECUTE AS SELF
AS
    DECLARE @entryTrancount int = @@trancount;

    BEGIN TRY
       --declare variable to use to tell whether to include the
       DECLARE @salesOrderIdChangedFlag bit =
                      CASE WHEN (SELECT salesLevelId
                                 FROM sales.contact
                                 WHERE contactId = @contactId) =
                                                          @salesLevelId
                           THEN 0 ELSE 1 END;

    DECLARE @query nvarchar(max);
    SET @query = '
    UPDATE sales.contact
    SET    firstName = ' + quoteName(@firstName,'''') + ',
           lastName = ' + quoteName(@lastName,'''') + ',
           companyName = ' + quoteName(@companyName, '''') + ',
           '+ case when @salesOrderIdChangedFlag = 1 then
           'salesLevelId = ' + quoteName(@salesLevelId, '''') + ',
           ' else '' end + 'personalNotes = ' + quoteName(@personalNotes, '''') + ',
           contactNotes = ' + quoteName(@contactNotes,'''') + '
      WHERE contactId = ' + cast(@contactId as varchar(10)) ;
      EXECUTE (@query);
    END TRY
    BEGIN CATCH
      IF @@trancount > 0
           ROLLBACK TRANSACTION;

      DECLARE @ERRORmessage nvarchar(4000)
      SET @ERRORmessage = 'Error occurred in procedure ''' +
                 OBJECT_NAME(@@procid) + ''', Original Message: '''
      + ERROR_MESSAGE() + '''';
      THROW 50000,@ERRORmessage,1;
    END CATCH
```

This is a pretty simple example, and as you can see the code is already getting pretty darn ugly. Of course, the advantage of encapsulation is still intact, since the user will be able to do exactly the same operation as before with no change to the public interface, but the code is immediately less manageable at the module level.

www.it-ebooks.info

An alternative you might consider would be an INSTEAD OF trigger to conditionally do the update on the column in question if the inserted and deleted columns don't match:

```
CREATE TRIGGER sales.contact$insteadOfUpdate
ON sales.contact
INSTEAD OF UPDATE
AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the rowcount
   --use inserted for insert or update trigger, deleted for update or delete trigger
   --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
   --that is equal to number of merged rows, not rows being checked in trigger
   DECLARE @msg varchar(2000), --used to hold the error message
   --use inserted for insert or update trigger, deleted for update or delete trigger
   --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
   --that is equal to number of merged rows, not rows being checked in trigger
          @rowsAffected int = (SELECT COUNT(*) FROM inserted);
   --          @rowsAffected int = (SELECT COUNT(*) FROM deleted);
   --no need to continue on if no rows affected
   IF @rowsAffected = 0 RETURN;

   BEGIN TRY
          --[validation blocks]
          --[modification blocks]
          --<perform action>

    UPDATE contact
    SET    firstName = inserted.firstName,
           lastName = inserted.lastName,
           companyName = inserted.companyName,
           personalNotes = inserted.personalNotes,
           contactNotes = inserted.contactNotes
    FROM   sales.contact AS contact
              JOIN inserted
                  on inserted.contactId = contact.contactId

    IF UPDATE(salesLevelId) --this column requires heavy validation
                            --only want to update if necessary
         UPDATE contact
         SET salesLevelId = inserted.salesLevelId
         FROM sales.contact AS contact
              JOIN inserted
                ON inserted.contactId = contact.contactId
        --this correlated subquery checks for rows that have changed
        WHERE EXISTS (SELECT *
                      FROM deleted
                      WHERE deleted.contactId =
                                    inserted.contactId
                        AND deleted. salesLevelId <>
                                    inserted. salesLevelId)
END TRY
```


www.it-ebooks.info

```
   BEGIN CATCH
            IF @@trancount > 0
                  ROLLBACK TRANSACTION;
           THROW;
   END CATCH
END;
```

This is a lot of code, but it's simple. This is one of the rare uses of `INSTEAD OF` triggers, but it's pretty simple to follow. Just update the simple columns and not the "high cost" columns, unless it has changed. The point of this is to note that the more encapsulated you get from the client, the more you can do in your code to modify the code to your needs. The stored procedure layer can be treated as a set of modules that return a set of data, save the state of some data, and so forth, without the client needing to know anything about the structure of anything other than the parameters and tabular data streams that are to be returned.

All Things Considered…What Do I Choose?

If the opinions in the previous two sections were not enough (and they weren't), this section lays out my opinions on what is good and bad about using ad hoc SQL and stored procedures. As Oscar Wilde was quoted as saying, "It is only about things that do not interest one that one can give a really unbiased opinion, which is no doubt the reason why an unbiased opinion is always absolutely valueless." This is a topic that I care about, and I have firm feelings about what is right and wrong. Of course, it is also true that many viable, profitable, and stable systems don't follow any of these opinions. That said, let's recap the pros and cons I have given for the different approaches. The pros of using ad hoc SQL are as follows:

- It gives a great deal of flexibility over the code, as the code can be generated right at runtime, based on metadata, or even the user's desires. The modification statement can only update column values that have changed.
- It can give adequate or even improved performance by only caching and parameterizing obviously matching queries. It also can be much easier to tailor queries in which you have wildly varying parameter and join needs.
- It's fast. If the programmer can write a SQL statement or use an API that does, there's less overhead learning about how to write stored procedures.

The cons are as follows:

- Your client code and database structures are tightly coupled, and when any little thing changes (column name, type, etc for example) in the database, it often requires making a change to the client code, requiring greater costs in deploying changes.
- Tying multiple statements together can be cumbersome, especially when transactions are required.
- API-generated queries often are not optimal, causing performance and especially maintenance issues when the database administrator has to optimize queries that cannot be easily modified.
- Performance-tuning database calls can be much harder to do, because modifying a statement, even to add a query hint, requires a recompile.

www.it-ebooks.info

For stored procedure access, the pros are as follows:

- The encapsulation of database code reduces what the user interface needs to know about the implemented database structures. If they need to change, often you can change the structures and tweak a procedure, and the client needn't know.
- You can easily manage security at the procedure level, with no need whatsoever to grant rights to base tables. This way, users don't have to have rights to any physical tables.
- You have the ability to do dynamic SQL in your procedures. In SQL Server 2005 and later, you can do this without the need to grant rights to objects using `EXECUTE AS`.
- Performance is improved, due to the parameterizing of all plans (unless otherwise specified).
- Performance tuning is made far simpler, due to the ability to tune a procedure without the client knowing the difference.

The cons for stored-procedure access are as follows:

- The rigid code of precompiled stored procedures can make coding them difficult.
- You can't effectively vary the columns affected by any T-SQL statement.
- There's a larger initial effort required to create the procedures.

With no outside influence other than this list of pros and cons and experience, I can state without hesitation that stored procedures are the way to go, if for no other reason other than the encapsulation angle. By separating the database code from the client code, you get an effective separation of data-manipulation code from presentation code. But "no outside influence" is a pipe dream, as developers will have their own ideas, and to be realistic, I am obviously open to the value of using an ORM type tool that encapsulates a lot of the work of application building as well, keeping in mind that development costs are dwarfed by maintenance costs when you consider a system outage means users aren't producing, but still getting paid.

Keep in mind that I'm not suggesting that all code that works with data should be in stored procedures. Too often when stored procedures are used as the complete data interface, the people doing the programming have a tendency to start putting all sorts of procedural code in the procedures, making them hard to write and hard to maintain. The next step is moaning that procedures are terrible, slow, and inflexible. This is often one of the sticking points between the two different opinions on how to do things. More or less, what's called for when building a user interface is to build stored procedures that replace T-SQL statements that you would have built in an ad hoc manner, using T-SQL control of flow language at a minimum. Several types of code act on data that shouldn't be in stored procedures or T-SQL:

- *Mutable business logic and rules*: T-SQL is a rigid language that can be difficult to work with. Even writing CLR SQL Server objects (covered in the next section of this chapter) is unwieldy in comparison to building an adequate business layer in your application.
- *Formatting data*: When you want to present a value in some format, it's best to leave this to the presentation layer or user interface of the application. You should use SQL Server primarily to do set-based operations using basic DML, and have as little of the T-SQL control of flow language as possible.

Probably the biggest drawback to using procedures in earlier versions of SQL Server was eliminated in 2005 in the `EXECUTE AS` clause on the procedure creation. By carefully using the `EXECUTE AS` clause, you can change the security context of the executor of a procedure when the ownership chain is broken. So, in any places where a dynamic call is needed, you can make a dynamic call, and it will look to the user exactly as a normal precompiled stored procedure would—again, making sure to avoid the very dangerous SQL Injection errors that are far too common.

www.it-ebooks.info

I don't want to sound as if any system largely based on ad hoc SQL is permanently flawed and just a festering pile of the smelly bits of an orangutan. Many systems have been built on letting the application handle all the code, especially when tools are built that need to run on multiple platforms. This kind of access is exactly what SQL-based servers were originally designed for, so it isn't going to hurt anything. At worst, you're simply not using one of the advanced features that SQL Server gives you in stored procedures.

The one thing that often tips the scales to using ad hoc access is time. The initial effort required to build stored procedures is going to be increased over just using ad hoc SQL generated from a mapping layer. In fact, for every system I've been involved with where our access plan was to use ad hoc SQL, the primary factor was time: "It takes too long to build the procedures," or "it takes too long to develop code to access the stored procedures." Or even, "The tool we are using doesn't support stored procedures." All this inevitably swings to the statement that "The DBA is being too rigid. Why do we want to…?"

These responses are a large part of why this section of the chapter needed to be written. It's never good to state that the ad hoc SQL is just plain wrong, because that's clearly not true. The issue is which is better, and stored procedures greatly tip the scale, at least until outside forces and developer talents are brought in.

T-SQL and the CLR

Many world-class and mission-critical corporate applications have been created using T-SQL and SQL Server, so why integrate SQL Server with the CLR? The fact is, integrating the CLR provides a host of benefits to developers and DBAs that wasn't possible or wasn't easy with SQL Server 2000 and earlier. It also opens up a plethora of questions about the applicability of this still reasonably new technology.

In the last two sections, I approached the topic of stored procedures and ad hoc access to data, but as of SQL Server 2005, there's another interesting architectural option to consider. Beyond using T-SQL to code objects, you can use a .NET language to write your objects to run not in interpreted manner that T-SQL objects do but rather in what is known as the SQLCLR, which is a SQL version of the CLR that is used as the platform for the .NET languages to build objects that can be leveraged by SQL Server just like T-SQL objects.

Using the SQLCLR, Microsoft provides a choice in how to program objects by using the enhanced programming architecture of the CLR for SQL Server objects. By hosting the CLR inside SQL Server, developers and DBAs can develop SQL Server objects using any .NET-compatible language, such as C# or Visual Basic. This opens up an entire new world of possibilities for programming SQL Server objects and makes the integration of the CLR one of the most powerful new development features of SQL Server.

Back when the CLR was introduced to us database types, it was probably the most feared new feature of SQL Server. As adoption of SQL Server versions 2005 and greater are almost completely the norm now, use of the CLR still may be the problem we suspected, but the fact is, properly built objects written in the CLR need to follow many of the same principals as T-SQL and the CLR can be very useful when you need it.

Microsoft chose to host the CLR inside SQL Server for many reasons; some of the most important motivations follow:

- *Rich language support*: .NET integration allows developers and DBAs to use any .NET—compatible language for coding SQL Server objects. This includes such popular languages as C# and VB.NET.
- *Complex procedural logic and computations*: T-SQL is great at set-based logic, but .NET languages are superior for procedural/iterative code. .NET languages have enhanced looping constructs that are more flexible and perform far better than T-SQL. You can more easily factor .NET code into functions, and it has much better error handling than T-SQL. T-SQL has some computational commands, but .NET has a much larger selection of computational commands. Most important for complex code, .NET ultimately compiles into native code while T-SQL is an interpreted language. This can result in huge performance wins for .NET code.

www.it-ebooks.info

- *String manipulation, complex statistical calculations, custom encryption, and so on*: As discussed earlier, heavy computational requirements such as string manipulation, complex statistical calculations, and custom encryption algorithms that don't use the native SQL Server encryption fare better with .NET than with T-SQL in terms of both performance and flexibility.
- *.NET Framework classes*: The .NET Framework provides a wealth of functionality within its many classes, including classes for data access, file access, registry access, network functions, XML, string manipulation, diagnostics, regular expressions, arrays, and encryption.
- *Leveraging existing skills*: Developers familiar with .NET can be productive immediately in coding SQL Server objects. Familiarity with languages such as C# and VB.NET, as well as being familiar with the .NET Framework is of great value. Microsoft has made the server-side data-access model in ADO.NET similar to the client-side model, using many of the same classes to ease the transition. This is a double-edged sword, as it's necessary to determine where using .NET inside SQL Server provides an advantage over using T-SQL. I'll consider this topic further throughout this section.
- *Easier and safer substitute for extended stored procedures*: You can write extended stored procedures in C++ to provide additional functionality to SQL Server. This ability necessitates an experienced developer fluent in C++ and able to handle the risk of writing code that can crash the SQL Server engine. Stored procedures written in .NET that extend SQL Server's functionality can operate in a managed environment, which eliminates the risk of code crashing SQL Server and allows developers to pick the .NET language with which they're most comfortable.
- *New SQL Server objects and functionality*: If you want to create user-defined aggregates or user-defined types (UDTs) that extend the SQL Server type system, .NET is your only choice. You can't create these objects with T-SQL. There's also some functionality only available to .NET code that allows for streaming table-valued functions.
- *Integration with Visual Studio*: Visual Studio is the premier development environment from Microsoft for developing .NET code. This environment has many productivity enhancements for developers. The Professional and higher versions also include a new SQL Server project, with code templates for developing SQL Server objects with .NET. These templates significantly ease the development of .NET SQL Server objects. Visual Studio .NET also makes it easier to debug and deploy .NET SQL Server objects.

However, while these are all good reasons for the concept of mixing the two platforms, it isn't as if the CLR objects and T-SQL objects are equivalent. As such, it is important to consider the reasons that you might choose the CLR over T-SQL, and vice versa, when building objects. The inclusion of the CLR inside SQL Server offers an excellent enabling technology that brings with it power, flexibility, and design choices. And of course, as we DBA types are cautious people, there's a concern that the CLR is unnecessary and will be misused by developers. Although any technology has the possibility of misuse, you shouldn't dismiss the SQLCLR without consideration as to where it can be leveraged as an effective tool to improve your database designs.

What really makes using the CLR for T-SQL objects is that in some cases, T-SQL just does not provide native access to the type of coding you need without looping and doing all sorts of machinations. In T-SQL it is the SQL queries and smooth handling of data that make it a wonderful language to work with. In almost every case, if you can fashion a SQL query to do the work you need, T-SQL will be your best bet. However, once you have to start

www.it-ebooks.info

using cursors and/or T-SQL control of flow language (for example, looping through the characters of a string or through rows in a table) performance will suffer mightily. This is because T-SQL is an interpreted language. In a well-thought-out T-SQL object, you may have a few non-SQL statements, variable declarations, and so on. Your statements will not execute as fast as they could in a CLR object, but the difference will often just be milliseconds if not microseconds.

The real difference comes if you start to perform looping operations, as the numbers of operations grow fast and really start to cost. In the CLR, the code is compiled and runs very fast. For example, I needed to get the maximum time that a row had been modified from a join of multiple tables. There were three ways to get that information. The first method is to issue a correlated subquery in the SELECT clause. I will demonstrate the query using several columns from the Sales Order Header and Customer tables in the AdventureWorks2012:

```
SELECT SalesOrderHeader.SalesOrderID,
               (SELECT MAX(DateValue)
                FROM (SELECT SalesOrderHeader.OrderDate AS DateValue
                      UNION ALL
                      SELECT SalesOrderHeader.DueDate AS DateValue
                      UNION ALL
                      SELECT SalesOrderHeader.ModifiedDate AS DateValue
                       UNION ALL
                       SELECT Customer.ModifiedDate as DateValue) AS dates
                                      ) AS lastUpdateTime
FROM  Sales.SalesOrderHeader
      JOIN Sales.Customer
           ON Customer.CustomerID = SalesOrderHeader.CustomerID;
```

Yes, Oracle users will probably note that this subquery performs the task that their GREATEST function will (or so I have been told many times). The approach in this query is a very good approach, and works adequately for most cases, but it is not necessarily the fastest way to answer the question that I've posed. A second, and a far more natural, approach for most programmers is to build a T-SQL scalar user-defined function:

```
CREATE FUNCTION dbo.date$getGreatest
(
    @date1 datetime,
    @date2 datetime,
    @date3 datetime = NULL,
    @date4 datetime = NULL
)
RETURNS datetime
AS
  BEGIN
    RETURN (SELECT MAX(dateValue)
            FROM ( SELECT @date1 AS dateValue
                          UNION ALL
                          SELECT @date2
                          UNION ALL
                          SELECT @date3
                          UNION ALL
                          SELECT @date4 ) AS dates);
  END;
```


www.it-ebooks.info

Now to use this, you can code the solution in the following manner:

```
SELECT SalesOrderHeader.SalesOrderID,
      dbo.date$getGreatest (SalesOrderHeader.OrderDate,
                            SalesOrderHeader.DueDate,
                            SalesOrderHeader.ModifiedDate,
                            Customer.ModifiedDate) AS lastUpdateTime
FROM   Sales.SalesOrderHeader
            JOIN Sales.Customer
                 ON Customer.CustomerID = SalesOrderHeader.CustomerID;
```

This is a pretty decent approach, though it is actually slower to execute than the native T-SQL approach in the tests I have run, as there is some overhead in using user-defined functions, and since the algorithm is the same, you are merely costing yourself. The third method is to employ a CLR user-defined function. The function I will create is pretty basic and uses what is really a brute force algorithm:

```
<SqlFunction(IsDeterministic:=True, DataAccess:=DataAccessKind.None, _
                                        Name:="date$getMax_CLR", _
                                        IsPrecise:=True)> _
Public Shared Function MaxDate(ByVal inputDate1 As SqlDateTime, _
                               ByVal inputDate2 As SqlDateTime, _
                               ByVal inputDate3 As SqlDateTime, _
                                ByVal inputDate4 As SqlDateTime _
                                       ) As SqlDateTime

       Dim outputDate As SqlDateTime

       If inputDate2 > inputDate1 Then outputDate = inputDate2
                                  Else outputDate = inputDate1
       If inputDate3 > outputDate Then outputDate = inputDate3
       If inputDate4 > outputDate Then outputDate = inputDate4

       Return New SqlDateTime(outputDate.Value)
   End Function
```

Generally, I just let VS .NET build and deploy the object into `tempdb` for me and script it out to distribute to other databases. (I have included this VB script in the downloads in a .vb file. If you want to build it yourself you will use SQL Data Tools. I have also included the T-SQL binary representations in the download for those who just want to build the object and execute it, though other than the binary representation it will seem very much like the T-SQL versions. In both cases, you will need to enable CLR using `sp_configure` setting '`clr enabled`').

For cases where the number of data parameters is great (ten or so in my testing on moderate, enterprise-level hardware), the CLR version will execute several times faster than either of the other versions. This is very true in most cases where you have to do some very functional-like logic, rather than using set-based logic.

After deploying, you the call you make still looks just like normal T-SQL (the first execution may take a bit longer due to the just in time compiler needing to compile the binaries the first time):

```
SELECT SalesOrderHeader.SalesOrderID,
      dbo.date$getMax_CLR (SalesOrderHeader.OrderDate,
                           SalesOrderHeader.DueDate,
                           SalesOrderHeader.ModifiedDate,
                           Customer.ModifiedDate) as lastUpdateTime
FROM   Sales.SalesOrderHeader
            JOIN Sales.Customer
                 ON Customer.CustomerID = SalesOrderHeader.CustomerID;
```


www.it-ebooks.info

Ignoring for a moment the performance factors, some problems are just easier to solve using the CLR, and the solution is just as good, if not better than using T-SQL. For example, to get a value from a comma-delimited list in T-SQL requires either a looping operation or the use of techniques requiring a Numbers table (as introduced in Chapter 12). This technique is slightly difficult to follow and is too large to reproduce here as an illustration.

However, in .NET, getting a comma-delimited value from a list is a built-in operation:

```
Dim tokens() As String = Strings.Split(s.ToString(), delimiter.ToString(), _
                                                                    -1, CompareMethod.Text)
    'return string at array position specified by parameter
    If tokenNumber > 0 AndAlso tokens.Length >= tokenNumber.Value Then
    Return tokens(tokenNumber.Value - 1).Trim()
```

In this section, I have probably made the CLR implementation sound completely like sunshine and puppy dogs. For some usages, particularly functions that don't access data other than what is in parameters, it certainly can be that way. Sometimes, however, the sun burns you, and the puppy dog bites you and messes up your new carpet. The fact is, the CLR is not bad in and of itself but it must be treated with the respect it needs. It is definitely not a replacement for T-SQL. It is a complementary technology that can be used to help you do some of the things that T-SQL does not necessarily do well.

The basic thing to remember is that while the CLR offers some great value, T-SQL is the language on which most all of your objects should be based. A good practice is to continue writing your routines using T-SQL until you find that it is just too difficult or slow to get done using T-SQL; then try the CLR.

In the next two sections, I will cover the guidelines for choosing either T-SQL or the CLR.

Guidelines for Choosing T-SQL

Let's get one thing straight: T-SQL isn't going away anytime soon. On the contrary, it's being enhanced, along with the addition of the CLR. Much of the same code that you wrote today with T-SQL back in SQL Server 7 or 2000 is still best done the same way with SQL Server 2005, 2008, 2012, and most likely going on for many versions of SQL Server. If your routines primarily access data, I would first consider using T-SQL. The CLR is a complementary technology that will allow you to optimize some situations that could not be optimized well enough using T-SQL.

The exception to this guideline of using T-SQL for SQL Server routines that access data is if the routine contains a significant amount of conditional logic, looping constructs, and/or complex procedural code that isn't suited to set-based programming. What's a significant amount? You must review that on a case-by-case basis. It is also important to ask yourself, "Is this task even something that should be done in the data layer, or is the design perhaps suboptimal and a different application layer should be doing the work?"

If there are performance gains or the routine is much easier to code and maintain when using the CLR, it's worth considering that approach instead of T-SQL. T-SQL is the best tool for set-based logic and should be your first consideration if your routine calls for set-based functionality (which should be the case for most code you write). I suggest avoiding rewriting your T-SQL routines in the CLR unless there's a definite benefit. If you are rewriting routines, do so only after trying a T-SQL option and asking in the newsgroups and forums if there is a better way to do it. T-SQL is a very powerful language that can do amazing things if you understand it. But if you have loops or algorithms that can't be done easily, the CLR is there to get you compiled and ready to go.

Keep in mind that T-SQL is constantly being enhanced with a tremendous leap in functionality. In SQL Server 2012 they have added vastly improved windowing functions, query paging extensions, and quite a lot of new functions to handle tasks that are unnatural in relational code. In 2008, they added such features as `MERGE`, table parameters, and row constructors; and in 2005, we got CTEs (which gave us recursive queries), the ability to `PIVOT` data, new `TRY-CATCH` syntax for improved error handling, and other features that we can now take advantage of. If there are new T-SQL features you can use to make code faster, easier to write, and/or easier to maintain, you should consider this approach before trying to write the equivalent functionality in a CLR language.

www.it-ebooks.info

■ **Note** Truthfully, if T-SQL is used correctly with a well designed database, almost all of your code will fit nicely into T-SQL code with only a function or two possibly needing to be created using the CLR.

Guidelines for Choosing a CLR Object

The integration of the CLR is an enabling technology. It's not best suited for all occasions, but it has some advantages over T-SQL that merit consideration. As we've discussed, CLR objects compile to native code, and is better suited to complex logic and CPU-intensive code than T-SQL. One of the best scenarios for the CLR approach to code is writing scalar functions that don't need to access data. Typically, these will perform an order (or orders) of magnitude faster than their T-SQL counterparts. CLR user-defined functions can take advantage of the rich support of the .NET Framework, which includes optimized classes for functions such as string manipulation, regular expressions, and math functions. In addition to CLR scalar functions, streaming table-valued functions is another great use of the CLR. This allows you to expose arbitrary data structures—such as the file system or registry—as rowsets, and allows the query processor to understand the data.

The next two scenarios where the CLR can be useful are user-defined aggregates and CLR based UDTs. You can only write user-defined aggregates with .NET. They allow a developer to perform any aggregate such as `SUM` or `COUNT` that SQL Server doesn't already do. Complex statistical aggregations would be a good example. I've already discussed .NET UDTs. These have a definite benefit when used to extend the type system with additional primitives such as point, SSN, and date (without time) types. As I discussed in Chapter 6, you shouldn't use .NET UDTs to define business objects in the database.

CLR Object Types

This section provides a brief discussion of each of the different types of objects you can create with the CLR. You'll also find additional discussion about the merits (or disadvantages) of using the CLR for each type.

You can build any of the following types of objects using the CLR:

- User-defined functions
- Stored procedures
- Triggers
- User-defined aggregates
- User-defined types

CLR User-Defined Functions

When the CLR was added to SQL Server, using it would have been worth the effort had it allowed you only to implement user-defined functions. Scalar user-defined functions that are highly computational are the sweet spot of coding SQL Server objects with CLR, particularly when you have more than a statement or two executing such functions. In fact, functions are the only type of objects that I have personally created and used using the CLR. I have seen some reasonable uses of several others, but they are generally fringe use cases. Those functions have been a tremendous tool for improving performance of several key portions of the systems I have worked with.

You can make both table value and scalar functions, and they will often be many times faster than corresponding T-SQL objects when there is no need for data other than what you pass in via parameters. CLR

www.it-ebooks.info

functions that have to query data via SQL become cumbersome to program, and usually coding them in T-SQL will simply be easier. Examples of functions that I have built using the CLR include:

- *Date functions*: Especially those for getting time zone information
- *Comparison functions*: For comparing several values to one another, like the `date$getMax` function shown earlier, though with more like thirty parameters instead of four.
- *Calculation functions*: Performing math functions or building a custom, soundex-style function for comparing strings and so forth.

Of all of the CLR object types, the user-defined functions are definitely the ones that you should consider using to replace your T-SQL objects, particularly when you are not interacting with data in SQL.

CLR Stored Procedures

As a replacement for extended stored procedures, .NET stored procedures provide a safe and relatively easy means to extend the functionality of SQL Server. Examples of extended stored procedures include `xp_sendmail`, `xp_cmdshell`, and `xp_regread`. Traditionally, extended stored procedures required writing code in a language such as C++ and ran in-process with the `sqlservr.exe` process. CLR stored procedures have the same capabilities as traditional, extended stored procedures, but they're easier to code and run in the safer environment of managed code. Of course, the more external resources that you access, the less safe your CLR object will be. The fact is that extended stored procedures were very often a dangerous choice, no matter how many resources you used outside of SQL server.

Another means of extending SQL Server was using the extended stored procedures beginning with `sp_OA%`. These used OLE automation to allow you to access COM objects that could be used to do things that T-SQL was unable to accomplish on its own. These objects were always slow and often unreliable and the CLR is an admirable replacement for the `sp_OA%` procedures and COM objects. You can use the .NET Framework classes to perform the same functionality as COM objects can.

If you are iterating through a set (perhaps using a cursor) performing some row-wise logic, you might try using CLR and the `SqlDataReader` class as it can be faster than using cursors. That said, it's best to start with T-SQL for your stored procedures that access SQL Server data, since there is very little that you cannot do using setwise manipulations, especially using `ROW_NUMBER` and the other ranking functions.

If performance of the stored procedure becomes a factor and there's lots of *necessary* procedural and computational logic within the stored procedure, experimenting with rewriting the stored procedure using CLR might be worthwhile. Quite often, if you find yourself doing too much in T-SQL code, you might be doing too much in the data layer, and perhaps you should move the procedural and computational logic away from the database to the middle tier and use the T-SQL stored procedure(s) primarily for data-access routines.

CLR Triggers

If you choose to use triggers in your database architecture, they're almost the exclusive domain of T-SQL. I felt this when I first heard of CLR triggers, and I feel this now. There is really not a good mainstream scenario where CLR triggers are going to be better than a well written T-SQL trigger. For any complex validation that might be done in a trigger, a better option is probably to use a CLR scalar function and call it from within a T-SQL trigger. Perhaps if there's significant procedural logic in a trigger, .NET would be a better choice than T-SQL, but just hearing the phrase "significant procedural logic in a trigger" makes me shudder a bit.

Be careful when using complex triggers. Almost any complex activity in a trigger is probably a cause for moving to use some form of queue where the trigger pushes rows off to some layer that operates asynchronously from the current transaction.

www.it-ebooks.info

User-Defined Aggregates

User-defined aggregates are a feature that have been desired for many years, and in 2005, with the introduction of the CLR, we were finally able to create our own aggregate functions. SQL Server already includes the most commonly used aggregate functions such as `SUM`, `COUNT`, and `AVG`. There might be a time when your particular business needs a special type of aggregate, or you need a complex statistical aggregate not supplied out of the box by SQL Server. Aggregates you create are not limited to numeric values either. An aggregate that I've always wanted is one similar to Sybase's `List()` aggregate that concatenates a list of strings in a column, separating them by commas.

In the code download for this book, you will find the code for the following aggregate function, called `string$list` (I also include the assembly in the T-SQL code for those of you who are purists who think procedural code is for other people!). Once the function's code is compiled and loaded into SQL server, you can do something like the following:

```
SELECT dbo.string$list(name) as names
FROM   (VALUES('Name'),('Name2'),('Name3')) AS Names (name)
```

This will return:

```
---------------------------
Name, Name2, Name3
```

In informal testing, running code like this using a custom aggregate can give an order of magnitude performance improvement over the T-SQL alternatives of using XML or over any trick you can do with variables and concatenation. A good part of the reason that the CLR version runs faster is that the T-SQL version is going to run a query for each group of values desired, while the CLR version is simply aggregating the products returned from the single query, using techniques that are natural to the SQL engine.

Versions of Visual Studio 2005 Professional and higher have included a SQL Server project and template that includes the stub for the functions that must be included as part of the contract for coding a .NET user-defined aggregate. For each aggregate, you must implement the `Init`, `Accumulate`, `Merge`, and `Terminate` functions. Besides the obvious performance benefit, there's also the flexibility benefit of being able to use such an aggregate with any column of any table. That is definitely unlike the T-SQL options where you need to hard-code which column and table are being aggregated.

■ **Note** As of SQL Server 2008, you can pass more than one parameter to your aggregate function, allowing you to perform far more interesting aggregations involving multiple values from each individual row returned by a query.

CLR User-Defined Types

Microsoft has built several types into SQL Server 2008 using the CLR. These are the `hierarchyId` and the spatial types. The fact is, though, that you should hesitate to base all of your datatypes on CLR types. The intrinsic, built-in datatypes will suffice for nearly every situation you will find, but if you have a need for a richer type with some complex handling, they are definitely available.

The largest downside to the CLR UDTs is that in order to get the full experience using the types, you will need to have the client set up to use them. To access the properties and methods of a UDT on the client and take full advantage of the new datatype, each client must have a copy of the UDT available in an assembly accessible by the client. If the code for a UDT is updated on SQL Server, the UDT class that's registered on each client that makes use of the UDT should be kept in sync with the server version to avoid any data problems. If the client does

www.it-ebooks.info

not have the UDT class available (like when you return the value of a column based on a UDT in Management Studio), the value will be returned as a hexadecimal value, unless you use the `.ToString` method on the type (which is a requirement for building a type). All in all, I generally steer clear of user defined types in pretty much all cases, because the management costs is typically way higher than the payoff.

Best Practices

The first half of the chapter discussed the two primary methods of architecting a SQL Server application, either by using stored procedures as the primary interface, or by using ad hoc calls built outside the server. Either is acceptable, but in my opinion the best way to go is to use stored procedures as much as possible. There are a few reasons:

- As precompiled batches of SQL statements that are known at design and implementation time, you get a great interface to the database that encapsulates the details of the database from the caller.
- They can be a performance boost, primarily because tuning is on a known set of queries, and not just on any query that the programmer might have written that slips by untested (not even maliciously; it could just be a bit of functionality that only gets used "occasionally").
- They allow you to define a consistent interface for security that lets you give users access to a table in one situation but not in another. Plus, if procedures are consistently named, giving access to database resources is far easier.

However, not every system is written using stored procedures. Ad hoc access can serve to build a fine system as well. You certainly can build a flexible architecture, but it can also lead to harder-to-maintain code that ends up with the client tools being tightly coupled with the database structures. At the very least, if you balk at the use of procedures, make sure to architect in a manner that makes tuning your queries reasonable without full regression testing of the application.

I wish I could give you definitive best practices, but there are so many possibilities, and either method has pros and cons. (Plus, there would be a mob with torches and pitchforks at my door, no matter how I said things must be done.) This topic will continue to be hotly contested, and rightly so. In each of the last few releases of SQL Server, Microsoft has continued to improve the use of ad hoc SQL, but it's still considered a best practice to use stored procedures if you can. I realize that in a large percentage of systems that are created, stored procedures are only used when there's a compelling reason to do so (like some complex SQL, or perhaps to batch together statements for a transaction).

Whether or not you decide to use stored-procedure access or use ad hoc calls instead, you'll probably want to code some objects for use in the database. Introduced in SQL Server 2005, there's another interesting decision to make regarding what language and technology to use when building several of these database objects. The best practices for the CLR usage are a bit more clear-cut:

- *User-defined functions*: When there's no data access, the CLR is almost always a better way to build user-defined functions; though the management issues usually make them a last choice unless there is a pressing performance issue. When data access is required, it will be dependent on the types of operations being done in the function, but most data access functions would be best at least done initially in T-SQL.
- *Stored procedures*: For typical data-oriented stored procedures, T-SQL is usually the best course of action. On the other hand, when using the CLR, it's far easier and much safer to create replacements for extended stored procedures (procedures typically named `xp_`) that do more than simply touch data.

www.it-ebooks.info

- *User-defined types*: For the most part, the advice here is to avoid them, unless you have a compelling reason to use them. For example, you might need complex datatypes that have operations defined between them (such as calculating the distance between two points) that can be encapsulated into the type. The client needs the datatype installed to get a natural interface; otherwise the clunky .NET-like methods need to be used (they aren't SQL-like).
- *User-defined aggregates*: You can only create these types of objects using .NET. User-defined aggregates allow for some interesting capabilities for operating on groups of data, like the example of a string aggregation.
- *Triggers*: There seems to be little reason to use triggers built into a CLR language. Triggers are about data manipulation. Even with DDL triggers, the primary goal is usually to insert data into a table to make note of a situation.

Summary

In this chapter full of opinions, what's clear is that SQL Server has continued to increase the number of options for writing code that accesses data. I've covered two topics that you need to consider when architecting your relational database applications using SQL Server. Designing the structure of the database is (reasonably) easy enough. Follow the principles set out by normalization to ensure that you have limited, if any, redundancy of data and limited anomalies when you modify or create data. On the other hand, once you have the database architected from an internal standpoint, you have to write code to access this data, and this is where you have a couple of seemingly difficult choices.

The case for using stored procedures is compelling (at least to many SQL architects), but it isn't a definite. Many programmers use ad hoc T-SQL calls to access SQL Server (including those made from middleware tools), and this isn't ever likely to change completely. This topic is frequently debated in blogs, forums, newsgroups, and church picnics with little budge from either side. I strongly suggest stored procedures for the reasons laid out in this chapter, but I do concede that it isn't the only way.

I then introduced the CLR features, and presented a few examples and even more opinions about how and when to use them. I dare say that some of the opinions concerning the CLR in this chapter might shift a little over time, but so far, it remains the case that the CLR is going to be most valuable as a tool to supercharge parts of queries, especially in places where T-SQL was poor because it was interpreted at runtime, rather than compiled. Usually this isn't a problem, because decent T-SQL usually has few procedural statements, and all the real work is done in set-based SQL statements.

The primary thing to take from this chapter is that lots of tools are provided to access the data in your databases. Use them wisely, and your results will be excellent. Use them poorly, and your results will be poor. Hopefully this advice will be of value to you, but as you were warned at the start of the chapter, a good amount of this chapter was opinion.

Last, the decision about the data-access method (i.e., ad hoc SQL code versus stored procedures and how much to use the CLR) should be chosen for a given project up front, when considering high-level design. For the sake of consistency, I would hope that the decision would be enforced across all components of the application(s). Nothing is worse than having to figure out the application as you dig into it.

■ **Note** A resource that I want to point out for further reading after this chapter is by Erland Sommarskog. His web site (`www.sommarskog.se`) contains a plethora of information regarding many of the topics I have covered in this chapter—and in far deeper detail. I would consider most of what I have said the introductory-level course, while his papers are nearly graduate-level courses in the topics he covers.

---

www.it-ebooks.info

CHAPTER 14

Reporting Design

Great questions make great reporting.

—Diane Sawyer

People use reporting in every aspect of their daily lives. Reports provide the local weather, the status of the morning train, and even the best place to get a cup of joe. And this is all before getting to work! At work, reports include enterprise dashboards, system outage reports, and timesheet cards. Any way you slice it, reporting is something that no one can get away from.

Wouldn't it be nice if you could understand how that reporting works? How the reports get put together and how the information is stored underneath the covers? This chapter will show you how that happens, by discussing two types of reporting you may encounter: analytical and aggregation. You will learn more about each area, including the types of dimensional modeling and how to get started on your own dimensional model. You will also create an aggregation model. Finally, you will look at common queries that can be used to retrieve information from each of these models.

Keep in mind that there are entire sets of books dedicated just to reporting, so you won't learn everything here, but hopefully, you'll learn enough to get you started on your own reporting solution.

Reporting Styles

Reporting can mean different things to different people and can be used in different ways. Reporting can lean toward the analytical, or even be used as an aggregation engine to display a specific set of information. Although each of these purposes is a valid reporting option, each has a unique way of storing information to ensure it can quickly and accurately satisfy its purpose and provide the correct information.

I've often had the debate with coworkers, clients, and friends of the best way to store data in a database. While methodologies differ, the overarching viewpoints revolve around two focuses: data in and data out. Or as Claudia Imhoff has stated in "Are You an Inny or an Outty?" (http://www.information-management.com/issues/19990901/1372-1.html). *Innies* are people who are most interested in storing the data into the table in the most efficient and quick way. On the other hand, *outties* are those who are most interested in making the output of the data as simple as possible for the end user. Both types of people can be involved in relational or data warehouse design. I unabashedly admit to being an outty. My entire goal of storing data is to make it easier for someone to get it out. I don't care if I need to repeat values, break normalization practices, or create descriptive column names that could compete with an anaconda for length.

www.it-ebooks.info

The reporting styles that I utilize to handle these reporting situations that are discussed in this chapter are:

- Analytical
- Aggregation

Each modeling paradigm has its own particular way of modeling the underlying database and storing the data. We will dig further into each of these reporting styles in this chapter.

Analytical Reporting

Probably the best-known method for storing data for analytical reporting is a dimensional model. *Analytics* focuses on two areas: understanding what happened and forecasting or trending what will happen. Both of these areas and questions can be solved using a dimensional model.

Dimensional modeling is a very business focused type of modeling. Knowing the business process of how the data is used is important in being able to model it correctly. The same set of information can actually be modeled in two different ways, based on how the information is used! In the database modeling technique of dimensional modeling, denormalization is a good thing. You will see how to create a dimensional model later in this chapter.

Dimensional models are typically used in data warehouses or datamarts. There are two leading methodologies for creating a dimensional model, led by two amazing technologists: Ralph Kimball and Bill Inmon. There are others as well, but we will focus on these two methodologies. While I typically use an approach more like Kimball's, both methodologies have benefits and can be used to create a successful dimensional model and analytical reporting solution.

Ralph Kimball

Ralph Kimball's approach to data warehousing and dimensional modeling is typically known as a *bottom-up approach*. This name signifies a bus architecture, in which a number of datamarts are combined using similar dimensions to create an enterprise data warehouse. Each data mart (and in turn, the data warehouse) is created used a star or snowflake schema, which contains one or more fact tables, linked to many dimensions.

To create the bus architecture, datamarts are created that are subject oriented and contain business logic for a particular department. Over time, new datamarts are created, using some of the same entities or dimensions as the original datamart. Eventually, the datamarts grow to create a full enterprise data warehouse.

For more information on Ralph Kimball's approach, see http://www.kimballgroup.com.

Bill Inmon

The other leading data warehouse methodology is proponed by Bill Inmon and is typically known as a *top-down approach*. In this scenario, "top-down" means starting with an enterprise warehouse and creating subject area datamarts from that data warehouse. The enterprise data warehouse is typically in a third-normal form, which was covered in Chapter 5, while data marts are typically in a dimensional format.

Inmon also created the concept of a corporate information factory, which combines all organization systems, including applications and data storage, into one cohesive machine. Operational data stores, enterprise data warehouses, and data management systems are prevalent in these systems.

See http://www.inmoncif.com for additional information on Bill Inmon's approach.

www.it-ebooks.info

Aggregation Reporting

A hybrid of analytical and relational reporting, *aggregation reporting* combines the best of both worlds to create a high-performing set of information in specific reporting scenarios. This information is used for some analysis and is also used operationally. By creating a set of aggregated values, report writers can quickly pull exactly the information they need.

Modeling summary tables typically occurs when a report that used an aggregated value becomes too slow for the end users. This could be because the amount of data that was aggregated has increased or the end users are doing something that wasn't originally planned for. To fix the slow reporting situation, summary tables can be created to speed up the reports without having to create a whole data warehouse.

Some scenarios where it may make sense to create summary tables include rolling up data over time and looking at departments or groups within a company where you can report on multiple levels. Aggregation reporting is especially useful when there is a separate reporting database where its resources can be consumed by queries and reports.

Requirements-Gathering Process

Before starting to do any modeling, it is essential to gather the requirements for the end solution. In this case, the word "requirements" doesn't mean how the end solution should look or who should have access to it. "Requirements" refers to who will use information and how they use it, particularly the business processes and results of using the data.

By talking to numerous members of the business, executive team, and key stakeholders, you will learn how the business works and what is valued. The corporate strategy and goals provide an important aspect to your requirements. They will ensure you are on the same page as the company and that your analytics can contribute to the high-level goals of the organization. At this time, I like to review the existing technical processes and data model. Because you understand the business inside and out, you will be able to see how the current infrastructure does or does not support the business.

Here, my system for gathering requirements veers off from typical processes. I like to create the initial data model as part of the requirements process. I find that I dig more deeply into understanding the relationship between entities, attributes, and processes if I am trying to put it into a data model. Then, if I can explain the data model in terms of a business process to the business folk, I am golden!

A recap of the steps listed in this section is shown in Figure 14-1.

Whether you require an analytical or aggregation reporting solution, you will still follow a similar requirements process. The process that ports the business process requirements into a particular type of data model (described as step 4) will be described in the following sections of this chapter.

The requirements-gathering phase is a very important step in any software development life cycle, including a reporting solution. Be firm about talking to the business as well as the technology department.

At the end of a business intelligence project, I often find myself in a situation where I am explaining a piece of the business to someone from another department. Because of the perpetual silos that occur in a company, it's too easy to ignore something that doesn't affect you directly. Your goal as a reporting modeler is to learn about all areas and processes. Often, you'll discover ties between the siloed departments that were impossible to initially see!

www.it-ebooks.info



Figure 14-1. *Requirements gathering steps diagram*

Dimensional Modeling for Analytical Reporting

The design principles in this section describe dimensional modeling, which is used for analytical reporting. Dimensional modeling takes a business process and separates it into logical units, typically described as entities, attributes, and metrics. These logical units are split into separate tables, called dimensions and facts. Additional tables can be included, but these are the two most common and important tables that are used.

Following are the different types of tables we will work with in a dimensional model:

- *Dimension:* A dimension table contains information about an entity. All descriptors of that entity are included in the dimension. The most common dimension is the date dimension, where the entity is a day and the descriptors include month, year, and day of week.
- *Fact:* A fact table is the intersection of all of the dimensions and includes the numbers you care about, also known as *measures*. The measures are usually aggregatable, so they can be summed, averaged, or used in calculations. An example of a fact table would be a store sales fact, where the measures include a dollar amount and unit quantity.
- *Bridge:* A bridge table, also known as a *many-to-many table*, links two tables together that cannot be linked by a single key. There are multiple ways of presenting this bridge, but the outcome is to create a many-to-many relationship between two dimensions tables.

We will start by describing the different types of dimensions that you can create and wrap up with the types of facts you can create. After we complete our discussion of facts and dimensions, we will end up with a complete health care insurer data model, as shown in Figure 14-2.

www.it-ebooks.info



Figure 14-2. *Completed health care payer dimensional model*

www.it-ebooks.info

Dimensions

As previously stated, a dimension is the table that describes an entity. Once the business process is described, the first step is to determine what dimensions we need. Then, we can decide on what type of dimension we need to use, what the *grain* (or smallest detail) of the dimension should be, and finally, what we want to put into the dimension. Let's start with business process.

While the claim payment business process is fairly complicated, we will use a limited version to illustrate a dimensional model. When an insurance company receives a claim for a member, the claim goes through an adjudication process. The claim can either be automatically adjudicated, manually adjudicated, or denied. As an insurance company, we need to know whether we adjudicated the claim, how much the physician requested based on the member's diagnosis, and how much we are willing to pay for that procedure.

Sometimes, translation between the technical and business folks can be difficult, so try to understand how the business works and talk in their language. Don't try to relate the business to another field, and don't assume that you understand what they are saying. Ask the questions in different ways and try to come up with edge cases to see if they may help you gain a deeper understanding. If 99 percent of camouflage pants come in green or gray, but 1 percent comes in purple, you must know about that 1 percent from a data perspective.

Once you've thought through the business process, it is easy to pick out the key entities. While getting started, it may help to write down the business as in the previous paragraph and then highlight the entities as you come to them. The following paragraph illustrates how to highlight the entities from the business process:

When an insurance company receives a claim for a *member*, the claim goes through an *adjudication process*. The claim can either be automatically adjudicated, manually adjudicated, or denied. As an insurance company, we need to know whether we adjudicated the claim, how much the *physician* requested based on the member's *diagnosis*, and how much we are willing to pay for that *procedure*.

Using those italicized phrases, our dimensions are: date, member, adjudication type, physician/provider, diagnosis, and procedure. Once we have the dimensions, it is important to talk to the business to find out what the grain for each of those dimensions should be. Do we receive claims on a daily or monthly basis? Is a member defined as an insurance number, a social security number, or a household? And so on. These are questions that must be asked of and answered by the business.

You must know where the data will be sourced from to ensure that the correct names and datatypes are used in the model. Fortunately, healthcare is a field that has standard definitions and files for many of the entities that we are interested in! During the walkthrough of each entity, any standard sources will be called out.

Once you have gone through the requirements process, discovered all of the business processes, and know everything there is to know about your business, it is time to start modeling! I like to start with dimensions, as the facts seem to fall into place after that. If you recall from earlier in the chapter, a dimension is an entity and its descriptors. Dimensions also include hierarchies, which are levels of properties that relate to each other in groups. Let's walk through a few examples of dimensions to solidify everything we've discussed.

Date Dimension

The date dimension is the most common dimension, as almost every business wants to see how things change over time. Sometimes, the business even wants to see information at a lower granularity than date, such as at an hour or minute level. While it is tempting to combine the time granularity with the date dimension, *don't do it*! Analysis is typically done at either the time or date level, so a rollup is not necessary, and you will inflate the number of rows in your dimension (365 days * 1440 minutes = 525,600 rows in your table). Let's walk through the thought process of creating the date dimension, and then, you can use a similar thought process to create a separate time dimension.

www.it-ebooks.info

You can always start with a base date dimension and modify it to suit your needs. The base date dimension should have the following features:

- *An integer key that uniquely identifies that row*: Sometimes, a surrogate key is used, which has absolutely no business value and can be implemented using an `IDENTITY` column. My recommendation is to use a smart key, which combines the year and date in an eight-digit number, such as `20110928`, for September 28, 2011 for the United States folks. Smart keys make sense for date and time dimensions, because you know the entity itself will never change. In other words, September 28, 2011 will always be in the month of September and will always be in the year 2011.
- *The business key that represents the entity*: For a date dimension, this value is very simple; it is just the date. This business key should always be the lowest granularity of information stored in your dimension.
- *Additional standard attributes*: These attributes can include information about whether this date is a holiday, whether a promotion is running during this time, or additional fields describing a fiscal calendar

The code to create a base date dimension table is listed here:

```
-- Create schema for all dimension tables
CREATE SCHEMA dim
GO

-- Create Date Dimension
CREATE TABLE dim.Date
(
    DateKey INTEGER NOT NULL,
    DateValue DATE NOT NULL,
    DayValue INTEGER NOT NULL,
    WeekValue INTEGER NOT NULL,
    MonthValue INTEGER NOT NULL,
    YearValue INTEGER NOT NULL
CONSTRAINT PK_Date PRIMARY KEY CLUSTERED
(
    DateKey ASC
))
GO
```

■ **Note** When I create my table and column names, I use a particular naming convention that includes schemas to describe the table type, capital letters for each word in the name, and a set of suffixes that describe the column type. While this standard works for me, you should use a convention that fits into your organization's standards.

To populate the dimension, you can use a stored procedure or script that will automatically add additional days as needed. Such a stored procedure follows. Note the initial insert statement that adds the unknown row if it doesn't exist.

www.it-ebooks.info

Unknown rows are used to symbolize that the relationship between a metric and a particular dimension entity does not exist. The relationship may not exist for a number of reasons:

- The relationship does not exist in the business process.
- The incoming feed does not yet know what the relationship should be.
- The information does not appear to be valid or applicable to the scenario.

By tying the fact table to the unknown row, it is very simple to see where the relationship is missing. The surrogate key for each unknown row is –1, and the descriptors contain variations of "unknown," such as UN, -1, and UNK. Each dimension that we will create will contain an unknown row to highlight this use case.

In some cases, it may make sense to create multiple unknown rows to distinguish between the three reasons for having an unknown row. One reason why you may want to do this is if you have many late-arriving dimensions, where you are very concerned with the rows that have not yet been linked to a dimension. Another reason you may want to do this is if you have poor data quality, and it is important to distinguish between acceptable and unacceptable unknown rows.

The following stored procedure uses only one unknown row:

```
-- Create Date Dimension Load Stored Precedure
CREATE PROCEDURE dim.LoadDate (@startDate DATETIME, @endDate DATETIME)
AS
BEGIN

IF NOT EXISTS (SELECT * FROM dim.Date WHERE DateKey = -1)
BEGIN
INSERT INTO dim.Date
SELECT -1, '01/01/1900', -1, -1, -1, -1
END

WHILE @startdate <= @enddate
  BEGIN
  IF NOT EXISTS (SELECT * FROM dim.Date WHERE DateValue = @startdate)
  INSERT INTO dim.Date
  SELECT CONVERT(CHAR(8), @startdate, 112) AS DateKey
     ,@startdate  AS DateValue
     ,DAY(@startdate)  AS DayValue
     ,DATEPART(wk, @startdate) AS WeekValue
     ,MONTH(@startdate)  AS MonthValue
     ,YEAR(@startdate)  AS YearValue
  SET @startdate = DATEADD(dd, 1, @startdate)
  END
END
GO
```

The outcome of this stored procedure is to load any date values that are not yet in the date dimension. Running the following query will insert two years of date:

```
EXECUTE dim.LoadDate '01/01/2011', '12/31/2012'
GO
```

A sample of the data in the table is shown here:

www.it-ebooks.info

```
DateKey     DateValue     DayValue    WeekValue   MonthValue  YearValue
--------    ----------    --------    ---------   ----------  ---------
-1          1900-01-01    -1          -1          -1          -1
20111230    2011-12-30    30          53          12          2011
20111231    2011-12-31    31          53          12          2011
20120101    2012-01-01    1           1           1           2012
20120102    2012-01-02    2           1           1           2012
```

This dimension and stored procedure may be exactly what you need, and if so, move onto the next section on slowly changing dimensions. On the other hand, there's a good chance that you'll need to go further with this dimension. To start, many organizations have fiscal calendars. You will use the same base date dimension, but add additional columns for `FiscalDayValue`, `FiscalWeekValue`, and so on. Along the same vein, you may also have date-specific descriptors that should be included. These could be as simple as calling out the federal holidays or as complicated as highlighting the various sales periods throughout the year. To include any of those additional items, you'll want to modify the stored procedure to use the logic specific to your organization.

Slowly Changing Dimension

The date dimension is about as simple as a dimension can get. The information can be populated once or over time and is never modified. The rest of the dimensions won't be that simple. To begin, we need to discuss the concept of slowly changing dimensions. Here are the most common types of slowly changing dimensions:

- *Type 1, or changing*: A type 1 change overwrites the value that was previously assigned to that attribute. A type 1 change is typically used when keeping the original value is not important. Example attributes include product category and company name.
- *Type 2, or historical*: A type 2 change is more in-depth than a type 1 change. It tracks any changes that occurred to that attribute over time so seeing the original, final, and any intermediate values is possible. Example historical attributes in which you may be interested include customer state and employee manager.
- *Type 3, or hybrid*: Finally, a type 3 change is a hybrid of both the type 1 and type 2 changes. It keeps track of the intermediate and final values on every row. This allows reporting to compare the current value with the latest value for any intermediate value or to be able to see the alternate view, as though the change didn't happen. This approach is not often used, but one good example is comparing the average temperature for a location at that time versus the final average temperature over the life of the warehouse.

As this book has hammered over and over again, uniqueness is so very important when dealing with data. And I'm not going to contradict that now, except to say that it is possibly even *more* important when dealing with reporting. You need to determine the unique identifier, also known as the business key, for each entity. This key is how you will know how to link a dimension to a fact and how to create additional rows in the case of a slowly changing dimension.

In the case of a health insurance company, a member can change personal information on a regular basis, and seeing how that information tracks over time may be important. Let's walk through the steps of creating the member dimension.

As with every dimension, we begin by determining the business key for the entity. For an insured member, the business key is the insurance number. Additional attributes for personal information that are analyzed

www.it-ebooks.info

include first name, last name, primary care physician, county, state, and membership duration. Our initial dimension can be created using the following script:

```
-- Create the Member dimension table
CREATE TABLE dim.Member
(
    MemberKey  INTEGER NOT NULL IDENTITY(1,1),
    InsuranceNumber  VARCHAR(12) NOT NULL,
    FirstName  VARCHAR(50) NOT NULL,
    LastName  VARCHAR(50) NOT NULL,
    PrimaryCarePhysician VARCHAR(100) NOT NULL,
    County  VARCHAR(40) NOT NULL,
    StateCode  CHAR(2) NOT NULL,
    MembershipLength VARCHAR(15) NOT NULL
CONSTRAINT PK_Member PRIMARY KEY CLUSTERED
(
    MemberKey ASC
))
GO
```

For demonstration's sake, load the member dimension with the following INSERT script:

```
-- Load Member dimension table
SET IDENTITY_INSERT [dim].[Member] ON
GO
INSERT INTO [dim].[Member]
([MemberKey],[InsuranceNumber],[FirstName],[LastName],[PrimaryCarePhysician]
 ,[County],[StateCode],[MembershipLength])
VALUES
(-1, 'UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UN','UNKNOWN')
GO
SET IDENTITY_INSERT [dim].[Member] OFF
GO
INSERT INTO [dim].[Member]
([InsuranceNumber],[FirstName],[LastName],[PrimaryCarePhysician]
 ,[County],[StateCode],[MembershipLength])
VALUES
('IN438973','Brandi','Jones','Dr. Keiser & Associates','Henrico','VA','<1 year')
GO
INSERT INTO [dim].[Member]
([InsuranceNumber],[FirstName],[LastName],[PrimaryCarePhysician]
 ,[County],[StateCode],[MembershipLength])
VALUES
('IN958394','Neil','Gomez','Healthy Lifestyles','Henrico','VA','1-2 year')
GO
INSERT INTO [dim].[Member]
([InsuranceNumber],[FirstName],[LastName],[PrimaryCarePhysician]
 ,[County],[StateCode],[MembershipLength])
VALUES
```


www.it-ebooks.info

```
('IN3867910','Catherine','Patten','Dr. Jenny Stevens','Spotsylvania','VA','<1 year')
GO
```

Use the following query to retrieve the sample data that follows:

```
select MemberKey, InsuranceNumber, FirstName, LastName, PrimaryCarePhysician
from dim.Member
GO
```

Here are the results:

MemberKey	Insurance	FirstName	LastName	PrimaryCarePhysician
-1	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
1	IN438973	Brandi	Jones	Dr. Keiser & Associates
2	IN958394	Neil	Gomez	Healthy Lifestyles
3	IN3867910	Catherine	Patten	Dr. Jenny Stevens

What do we do when a piece of information changes? Well, if it's something that we want to track, we use a type 2 change. If it's something we don't care about, we use type 1. We must know which type we want to track at modeling time to make some adjustments. In the case of the member dimension, we want to use a type 2 attribute. We know this because we want to track if a member's primary care physician (PCP) changes to verify that we are only paying out for visits to a PCP. To track this information as the type 2 attribute, we need to add a flag called `isCurrent` to indicate which row contains the most current information. The code to add this field is listed here:

```
ALTER TABLE dim.Member
ADD isCurrent INTEGER NOT NULL DEFAULT 1
GO
```

When a load process runs to update the member dimension, it will check to see if there is a match on the business key of `InsuranceNumber`, and if so, it will add an additional row but flip the `isCurrent` bit on the old row to 0. Use the following code to emulate Brandi changing her primary care physician and updating the old record:

```
INSERT INTO [dim].[Member]
([InsuranceNumber],[FirstName],[LastName],[PrimaryCarePhysician]
 ,[County],[StateCode],[MembershipLength])
VALUES
('IN438973','Brandi','Jones','Dr. Jenny Stevens','Henrico','VA','<1 year')
GO

UPDATE [dim].[Member] SET isCurrent = 0
WHERE InsuranceNumber = 'IN438973' AND PrimaryCarePhysician = 'Dr. Keiser & Associates'
GO
```

If we use a similar query as before, but add the new `isCurrent` column as follows, we see the full change of the dimension:

```
select MemberKey, InsuranceNumber, FirstName, LastName, PrimaryCarePhysician, isCurrent
from dim.Member
```

And here are the results:

www.it-ebooks.info

```
MemberKey   Insurance   FirstName   LastName   PrimaryCarePhysician      isCurrent
---------   ---------   ---------   --------   ----------------------    ---------
-1          UNKNOWN     UNKNOWN     UNKNOWN    UNKNOWN                   1
1           IN438973    Brandi      Jones      Dr. Keiser & Associates   0
2           IN958394    Neil        Gomez      Healthy Lifestyles        1
3           IN3867910   Catherine   Patten     Dr. Jenny Stevens         1
4           IN438973    Brandi      Jones      Dr. Jenny Stevens         1
```

■ **Note** For a more detailed explanation of how to programmatically add the logic for a type 2 dimension using SSIS, see Apress's *Pro SQL Server 2012 Integration Services* by Michael Coles and Francis Rodrigues.

An alternative to using the `isCurrent` flag is to use two columns: start date and end date. The start date describes when the row first became active, and the end date is when that row has expired. If the row has not yet expired, the value is null. If you wanted to see all current values, you can query all rows where the end date is null. In addition to that information, you can also see when each dimension row changed without having to look at the fact table.

You may ask yourself why you need any flag or date range. Isn't it possible to just find the latest row based on the growing identity key, where the maximum value is the current one? You would be absolutely correct that you can get the same information; however, the query to pull that information is more complicated and difficult to pull. As a case in point, would you rather write the query in Listing 14-1 or 14-2 on a regular basis?

Listing 14-1. *Preferred Query to Pull the Latest Dimension Row*

```
select * from dim.Member where isCurrent = 1
GO
```

Listing 14-2. *Rejected Query to Pull the Latest Dimension Row*

```
select * from (
   select m.*, ROW_NUMBER() OVER (PARTITION BY m.InsuranceNumber
   ORDER BY m.MemberKey DESC) As Latest
   from dim.Member m ) LatestMembers
WHERE LatestMembers.Latest = 1
GO
```

The provider dimension is similar to the member dimension, as it can change over time. The information in this information is based on a standard through the National Provider Registry, which can be found at http://www.cms.gov/NationalProvIdentStand/downloads/Data_Dissemination_File-Readme.pdf. By using a limited set of information, we can create a dimension as described here:

```
-- Create the Provider dimension table
CREATE TABLE dim.Provider (
    ProviderKey INTEGER IDENTITY(1,1) NOT NULL,
    NPI VARCHAR(10) NOT NULL,
    EntityTypeCode INTEGER NOT NULL,
    EntityTypeDesc VARCHAR(12) NOT NULL, -- (1:Individual,2:Organization)
    OrganizationName VARCHAR(70) NOT NULL,
    DoingBusinessAsName VARCHAR(70) NOT NULL,
```


www.it-ebooks.info

```
    Street VARCHAR(55) NOT NULL,
    City VARCHAR(40) NOT NULL,
    State VARCHAR(40) NOT NULL,
    Zip VARCHAR(20) NOT NULL,
    Phone VARCHAR(20) NOT NULL,
    isCurrent INTEGER NOT NULL DEFAULT 1
 CONSTRAINT PK_Provider PRIMARY KEY CLUSTERED
(
    ProviderKey ASC
))
GO
```

Add an initial set of data using the following query:

```
-- Insert sample data into Provider dimension table
SET IDENTITY_INSERT [dim].[Provider] ON
GO
INSERT INTO [dim].[Provider]
([ProviderKey],[NPI],[EntityTypeCode],[EntityTypeDesc],[OrganizationName],
    [DoingBusinessAsName],[Street],[City],[State],[Zip],[Phone])
VALUES
(-1, 'UNKNOWN',-1,'UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOW
N')
GO
SET IDENTITY_INSERT [dim].[Provider] OFF
GO

INSERT INTO [dim].[Provider]
([NPI],[EntityTypeCode],[EntityTypeDesc],[OrganizationName],
   [DoingBusinessAsName],[Street],[City],[State],[Zip],[Phone])
VALUES
('1234567',1,'Individual','Patrick Lyons','Patrick Lyons','80 Park St.',
'Boston','Massachusetts','55555','555-123-1234')
GO
INSERT INTO [dim].[Provider]
([NPI],[EntityTypeCode],[EntityTypeDesc],[OrganizationName],
    [DoingBusinessAsName],[Street],[City],[State],[Zip],[Phone])
VALUES
('2345678',1,'Individual','Lianna White, LLC','Dr. White & Associates','74 West Pine Ave.',
'Waltham','Massachusetts','55542','555-123-0012')
GO
INSERT INTO [dim].[Provider]
([NPI],[EntityTypeCode],[EntityTypeDesc],[OrganizationName],
    [DoingBusinessAsName],[Street],[City],[State],[Zip],[Phone])
VALUES
('76543210',2,'Organization','Doctors Conglomerate, Inc','Family Doctors','25 Main Street Suite
108',
'Boston','Massachusetts','55555','555-321-4321')
GO
INSERT INTO [dim].[Provider]
([NPI],[EntityTypeCode],[EntityTypeDesc],[OrganizationName],
```


www.it-ebooks.info

```
    [DoingBusinessAsName],[Street],[City],[State],[Zip],[Phone])
VALUES
('3456789',1,'Individual','Dr. Drew Adams','Dr. Drew Adams','1207 Corporate Center',
'Peabody','Massachusetts','55554','555-234-1234')
GO
```

A dimension can have attributes of multiple types, so the last name can be type 1 while the primary care physician is type 2. I've never understood why they're called slowly changing dimensions when they exist at an attribute level, but that's the way it is.

Snowflake Dimension

In our discussion of modeling methodologies, we talked about star schemas and snowflake schemas. The key differentiator between those two methodologies is the snowflake dimension. A *snowflake dimension* takes the information from one dimension and splits it into two dimensions. The second dimension contains a subset of information from the first dimension and is linked to the first dimension, rather than the fact, through a surrogate key.

A great example of this is when one entity can be broken into two subentities that change on their own. By grouping the subentities' properties together in their own dimension, they can change on their own or be linked directly to the fact on its own, instead of using the intermediary dimension, if the grain applies. However, breaking out an entity into a snowflake dimension is not without its downfalls. Never, ever, ever link both the first and second dimension to the same fact table. You'd create a circular relationship that could potentially cause discrepancies if your loads are not kept up to date. I also recommend never snowflaking your dimensions to more than one level, if at all. The added complexity and maintenance needed to support such a structure is often not worthwhile when alternative methods can be found. Let's talk through a possible snowflake dimension, when you would want to use it, and alternative methods.

To continue our healthcare example, we will look at the types of insurance coverage provided by the company. There are multiple types of plans that each contains a set of benefits. Sometimes, we look at the metrics at the benefit level and other times, at the plan level. Even though we sometimes look at benefits and plans differently, there is a direct relationship between them.

Let's begin by looking at each of the tables separately. The benefit table creation script is listed here:

```
-- Create the Benefit dimension table
CREATE TABLE dim.Benefit(
    BenefitKey INTEGER IDENTITY(1,1) NOT NULL,
    BenefitCode INTEGER NOT NULL,
    BenefitName VARCHAR(35) NOT NULL,
    BenefitSubtype VARCHAR(20) NOT NULL,
    BenefitType VARCHAR(20) NOT NULL
CONSTRAINT PK_Benefit PRIMARY KEY CLUSTERED
(
    BenefitKey ASC
))
GO
```

The health plan creation script is shown here:

```
-- Create the Health Plan dimension table
CREATE TABLE dim.HealthPlan(
    HealthPlanKey INTEGER IDENTITY(1,1) NOT NULL,
    HealthPlanIdentifier CHAR(4) NOT NULL,
    HealthPlanName VARCHAR(35) NOT NULL
```


www.it-ebooks.info

```
CONSTRAINT PK_HealthPlan PRIMARY KEY CLUSTERED
(
    HealthPlanKey ASC
))
GO
```

I've already explained that a set of benefits can be tied to a plan, so we need to add a key from the benefit dimension to the plan dimension. Use an ALTER statement to do this, as shown:

```
ALTER TABLE dim.Benefit
ADD HealthPlanKey INTEGER
GO
ALTER TABLE dim.Benefit WITH CHECK
ADD CONSTRAINT FK_Benefit_HealthPlan
FOREIGN KEY(HealthPlanKey) REFERENCES dim.HealthPlan (HealthPlanKey)
GO
```

When the benefit key is added to a fact, the plan key does not also need to be added but will still be linked and can be used to aggregate or filter the information. In addition, if the grain of one of the fact tables is only at the plan level, the plan key can be used directly without having to use the benefit key.

We will not use the Benefit dimension in our model, but we will use the HealthPlan dimension. You can populate the data using the following script:

```
-- Insert sample data into Health plan dimension
INSERT INTO [dim].[HealthPlan]
 ([HealthPlanIdentifier],[HealthPlanName])
VALUES ('BRON','Bronze Plan')
GO
INSERT INTO [dim].[HealthPlan]
 ([HealthPlanIdentifier],[HealthPlanName])
VALUES ('SILV','Silver Plan')
GO
INSERT INTO [dim].[HealthPlan]
 ([HealthPlanIdentifier],[HealthPlanName])
VALUES ('GOLD','Gold Plan')
GO
```

An important concept that we have not yet covered is how to handle hierarchies in your data structure. Hierarchies are an important concept that you will hear about over and over again. Each level of the hierarchy is included in the same dimension, and the value of each parent level is repeated for every child value. In the Benefit dimension, BenefitSubtype and BenefitType create a hierarchy. Ensure that all subtypes have the same type in the load process to help later reporting.

Type Dimension

Another common dimension type that often appears is what I like to call a *type dimension*, because it is often simple to identify due to its name. This dimension is a very simple one that typically contains a few type 1 changing attributes. Not all entities have to contain the word "type," but it's a nice indicator. Examples include transaction type, location type, and sale type.

www.it-ebooks.info

In our health care example, the adjudication type is a perfect example of this dimension. The adjudication type dimension contains two columns and four rows, and the full creation script is shown here:

```
-- Create the AdjudicationType dimension table
CREATE TABLE dim.AdjudicationType (
    AdjudicationTypeKey INTEGER IDENTITY(1,1) NOT NULL,
    AdjudicationType VARCHAR(6) NOT NULL,
    AdjudicationCategory VARCHAR(8) NOT NULL
CONSTRAINT PK_AdjudicationType PRIMARY KEY CLUSTERED
(
    AdjudicationTypeKey ASC
))
GO
```

The values for the adjudication type dimension are static, so they can be loaded once, instead of on a recurring or nightly basis. The following `INSERT` script will insert the three business value rows and the one unknown row:

```
-- Insert values for the AdjudicationType dimension
SET IDENTITY_INSERT dim.AdjudicationType ON
GO
INSERT INTO dim.AdjudicationType
    (AdjudicationTypeKey, AdjudicationType, AdjudicationCategory)
VALUES (-1, 'UNKNWN', 'UNKNOWN')
INSERT INTO dim.AdjudicationType
    (AdjudicationTypeKey, AdjudicationType, AdjudicationCategory)
VALUES (1, 'AUTO', 'ACCEPTED')
INSERT INTO dim.AdjudicationType
    (AdjudicationTypeKey, AdjudicationType, AdjudicationCategory)
VALUES (2, 'MANUAL', 'ACCEPTED')
INSERT INTO dim.AdjudicationType
    (AdjudicationTypeKey, AdjudicationType, AdjudicationCategory)
VALUES (3, 'DENIED', 'DENIED')
GO
SET IDENTITY_INSERT dim.AdjudicationType OFF
GO
```

The values in this table can then be shown as follows:

```
AdjudicationTypeKey   AdjudicationType   AdjudicationCategory
-------------------   ----------------   --------------------
-1                    UNKNWN             UNKNOWN
1                     AUTO               ACCEPTED
2                     MANUAL             ACCEPTED
3                     DENIED             DENIED
```

This provides an easy way to filter any claim that you have on either the actual type of adjudication that occurred or the higher-level category.

Our final two dimensions also fit into this category: `Diagnosis` and `HCPCSProcedure`. Diagnosis and procedure codes contain a standard set of data fields. Diagnoses and procedure codes are known as International Classification of Diseases, Tenth Revision, Clinical Modification (ICD10-CM) and Healthcare Common

www.it-ebooks.info

Procedure Coding System (HCPCS) codes, respectively. We can access this set of information through the Centers for Medicare and Medicaid Services web site (http://www.cms.gov). The creation script for these dimensions follows:

```
-- Create Diagnosis dimension table
CREATE TABLE dim.Diagnosis(
    DiagnosisKey int IDENTITY(1,1) NOT NULL,
    DiagnosisCode char(7) NULL,
    ShortDesc varchar(60) NULL,
    LongDesc varchar(322) NULL,
    OrderNumber int NULL,
 CONSTRAINT PK_Diagnosis PRIMARY KEY CLUSTERED
(
    DiagnosisKey ASC
))
GO

-- Create HCPCSProcedure dimension table
CREATE TABLE dim.HCPCSProcedure (
    ProcedureKey INTEGER IDENTITY(1,1) NOT NULL,
    ProcedureCode CHAR(5) NOT NULL,
    ShortDesc VARCHAR(28) NOT NULL,
    LongDesc VARCHAR(80) NOT NULL
 CONSTRAINT PK_HCPCSProcedure PRIMARY KEY CLUSTERED
(
    ProcedureKey ASC
))
GO
```

Depending on how in-depth your business needs are, you can add more fields to the diagnosis and procedure dimensions. For a full list of all available diagnosis fields, see http://www.cms.gov/ICD10/downloads/ICD10OrderFiles.zip, and for all available procedure fields, see https://www.cms.gov/HCPCSReleaseCodeSets/Downloads/contr_recordlayout_2011.pdf. You can also use the files provided to load the set of static rows (for each year that the code set is released). Sample Integration Services packages are provided with the code for this chapter to load these values.

Facts

Now that we've looked at the entities and properties that describe our business process, we need to tie it all together with the metrics. The fact table combines all dimensions and the metrics associated with our business process. We can create many types of facts, just as we can create many types of dimensions. We will discuss two of the more commonly occurring fact types now: transaction and snapshot.

Transaction Fact

The fact table that most people know is the *transaction fact*. This table relates the metrics and dimensions together very simply: every action that occurs in the business process, there is one row in the table. It is important to understand the grain of the table, which is the combination of the dimensions that describes what the metric is used for. For example, if you were to look at a bank transaction, the grain of the table could consist of a day, bank location, bank teller, customer, and transaction type. This is in comparison to a bank promotion where the grain could be hour, bank location, customer, and promotion type.

www.it-ebooks.info

Along with the dimension grain, metrics make up the fact table. The metrics are typically numeric values, such as currencies, quantities, or counts. These values can be aggregated in some fashion: by summing, averaging, or using the maximum or minimum values.

To continue our healthcare example, one transaction fact would be the payouts for insurance claims. Since we've already created our dimensions, there is a good chance we already know what our grain should be, but let's walk through the business process to make sure we have everything we need.

For this example, we have talked to the business owners and understand the business process down to the smallest detail. We know that our grain is day, insurance number, adjudication value, ICD-10 code, and HCPC. Once we have the grain, we are very close to having a complete transaction fact.

Now that we have the dimensional grain of the fact, the next step is to describe the metrics related to those dimensions. We will use the same process we followed to determine our dimensions. Look at the business process description that started our dimension discussion, and determine the aggregatable values that become the metrics, and pull out the desired values. An example of this process with metrics italicized can be seen in the following paragraph:

When an insurance company receives a claim for a member, the claim goes through an adjudication process. The claim can either be automatically adjudicated, manually adjudicated, or denied. As an insurance company, we need to know *whether we* adjudicated the *claim*, *how much the physician requested* based on the member's diagnosis, and *how much we are willing to pay* for that procedure.

Based on the italicized phrases, the desired metrics are adjudication counts, claim amount, and payout amount. It is important to look at how the business uses these metrics. Existing reports are a great place to start, but you also want to talk to the business analysts to understand what they do with that information. If they always take the information from the report and then perform manipulations in Excel, you want to try to replicate the manipulations they are doing directly into your dimensional model.

Our completed fact table is shown here:

```
-- Create schema for all fact tables
CREATE SCHEMA fact
GO

-- Create Claim Payment transaction fact table
CREATE TABLE fact.ClaimPayment
(
    DateKey INTEGER NOT NULL,
    MemberKey INTEGER NOT NULL,
    AdjudicationTypeKey INTEGER NOT NULL,
    ProviderKey INTEGER NOT NULL,
    DiagnosisKey INTEGER NOT NULL,
    ProcedureKey INTEGER NOT NULL,
    ClaimID VARCHAR(8) NOT NULL,
    ClaimAmount DECIMAL(10,2) NOT NULL,
    AutoPayoutAmount DECIMAL(10,2) NOT NULL,
    ManualPayoutAmount DECIMAL(10,2) NOT NULL,
    AutoAdjudicatedCount INTEGER NOT NULL,
    ManualAdjudicatedCount INTEGER NOT NULL,
    DeniedCount INTEGER NOT NULL
)
GO
```

Of course, we need to link our fact table to the dimensions that we've already created, using foreign keys. When loading your data, it may make sense to disable or remove the constraints and enable them afterward to increase the speed of the load. I like to have the relationships after the fact to double check that the load of

www.it-ebooks.info

information was successful and that the fact doesn't reference dimensions keys that do not exist. To set up the foreign keys, use the following script:

```
-- Add foreign keys from ClaimPayment fact to dimensions
ALTER TABLE fact.ClaimPayment WITH CHECK
ADD CONSTRAINT FK_ClaimPayment_AdjudicationType
FOREIGN KEY(AdjudicationTypeKey) REFERENCES dim.AdjudicationType (AdjudicationTypeKey)
GO

ALTER TABLE fact.ClaimPayment WITH CHECK
ADD CONSTRAINT FK_ClaimPayment_Date
FOREIGN KEY(DateKey) REFERENCES dim.Date (DateKey)
GO

ALTER TABLE fact.ClaimPayment WITH CHECK
ADD CONSTRAINT FK_ClaimPayment_Diagnosis
FOREIGN KEY(DiagnosisKey) REFERENCES dim.Diagnosis (DiagnosisKey)
GO

ALTER TABLE fact.ClaimPayment WITH CHECK
ADD CONSTRAINT FK_ClaimPayment_HCPCSProcedure
FOREIGN KEY(ProcedureKey) REFERENCES dim.HCPCSProcedure (ProcedureKey)
GO

ALTER TABLE fact.ClaimPayment WITH CHECK
ADD CONSTRAINT FK_ClaimPayment_Member
FOREIGN KEY(MemberKey) REFERENCES dim.Member (MemberKey)
GO

ALTER TABLE fact.ClaimPayment WITH CHECK
ADD CONSTRAINT FK_ClaimPayment_Provider
FOREIGN KEY(ProviderKey) REFERENCES dim.Provider (ProviderKey)
GO
```

How did we come up with six metrics from the three underlined descriptions from our business process? In this case, we talked to the business owners and found out that they always look at payout amounts in a split fashion: either by the automatically adjudicated or manually adjudicated amounts. We will model this business scenario by splitting the value into two separate columns based on whether the adjudicated flag is set to `AUTO` or `MANUAL`. This means that one column will always have 0 for every row. While this approach may seem counterintuitive, it will help us to query the end result without having to always filter on the adjudication type dimension. Since we know they will always look at one or the other, consider the queries in Listings 14-3 and 14-4. Which would you rather have to write every time you pull payment information from the database?

Listing 14-3. *Preferred Query to Pull Amount Information*

```
select AutoPayoutAmount, ManualPayoutAmount
from fact.ClaimPayment
GO
```

Listing 14-4. *Rejected Query to Pull Amount Information*

```
select CASE dat.AdjudicationType
            WHEN 'AUTO'
            THEN fp.ClaimAmount
```


www.it-ebooks.info

```
            ELSE 0
            END  as AutoPayoutAmount
      , CASE dat.AdjudicationType
        WHEN 'MANUAL'
        THEN fp.ClaimAmount
        ELSE 0
        END as ManualPayoutAmount
from fact.ClaimPayment fp
left join dim.AdjudicationType dat on fp.AdjudicationTypeKey=dat.AdjudicationTypeKey
GO
```

The next metric we want to discuss deeper is the adjudication counts, which help the business in figuring out how many of their claims are adjudicated (manually versus automatically) and denied. Rather than force the business to use the adjudication flag again, we will add three columns for each adjudication type. This set of columns will have a value of 1 in one of the columns and 0 in the other two. Some people prefer to think of these as flag columns with bit columns that are flipped if the value fits that row, but I prefer to think of and use numeric values that are easily aggregated.

When creating your SQL Server tables, always use the smallest sized data type that you can. This is another scenario where understanding the business is essential. The data type should be large enough to handle any data value that comes in but small enough that it doesn't waste server space.

You can use a query similar to the following to create a set of random rows for your fact tables:

```
-- Insert sample data into ClaimPayment fact table
DECLARE @i INT
SET @i = 0
WHILE @i < 1000
BEGIN
INSERT INTO fact.ClaimPayment
(
    DateKey, MemberKey, AdjudicationTypeKey, ProviderKey, DiagnosisKey,
    ProcedureKey, ClaimID, ClaimAmount, AutoPayoutAmount, ManualPayoutAmount,
    AutoAdjudicatedCount, ManualAdjudicatedCount, DeniedCount
)
SELECT
    CONVERT(CHAR(8), DATEADD(dd, RAND() * -100, getdate()), 112),
    (SELECT CEILING((COUNT(*) - 1) * RAND()) from dim.Member),
    (SELECT CEILING((COUNT(*) - 1) * RAND()) from dim.AdjudicationType),
    (SELECT CEILING((COUNT(*) - 1) * RAND()) from dim.Provider),
    (SELECT CEILING((COUNT(*) - 1) * RAND()) from dim.Diagnosis),
    (SELECT CEILING((COUNT(*) - 1) * RAND()) from dim.HCPCSProcedure),
    'CL' + CAST(@i AS VARCHAR(6)),
    RAND() * 100000,
    RAND() * 100000 * (@i % 2),
    RAND() * 100000 * ((@i+1) % 2),
    @i % 2,
    (@i+1) % 2,
    0
SET @i = @i + 1
END
GO
```


www.it-ebooks.info

Finally, the remaining column in the claim payment fact to discuss is the `ClaimID`. This column is the odd man out, in that it looks like a dimensional attribute but lives in the fact. These columns are known as *degenerate dimensions* and are typically identifiers or transaction numbers. In scenarios where an entity has an almost one-to-one relationship with the fact, putting the entity in its own dimension would just waste space. These columns are typically used for more detailed or drill-down reporting, so we can include the data in the fact for ease of reporting.

Snapshot Fact

The snapshot fact is not always intuitive when thinking through a business process or trying to query your model, but it is extremely powerful. As the name suggests, the *snapshot fact* records a set of information as a snapshot in time. Typical snapshot facts include financial balances on a daily basis and monthly membership counts.

The healthcare data model we will complete in this section is shown in Figure 14-3.

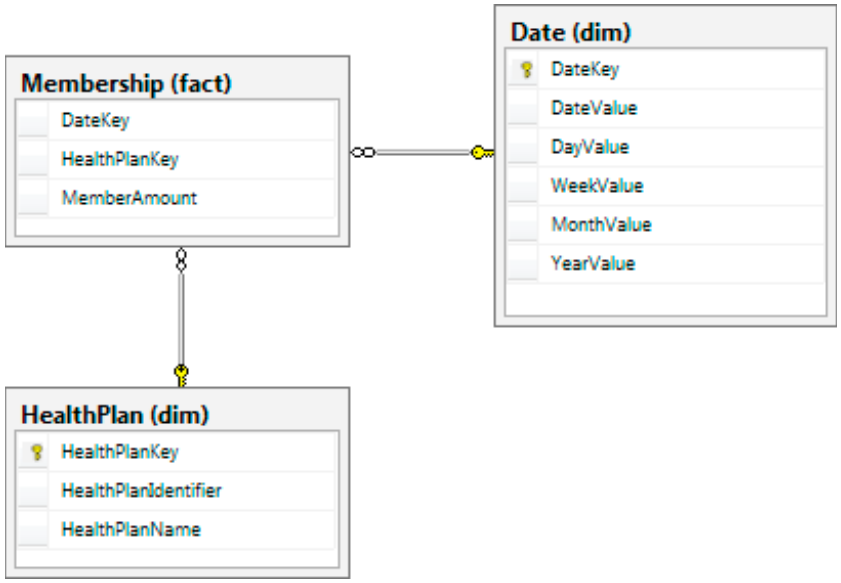


Figure 14-3. *Snapshot fact dimensional model for our health care payer*

In our healthcare example, daily member count is a perfect example of a snapshot fact. We begin by following a similar process to determine our dimensions and metrics. Our business process is listed below with dimensions in bold and metrics italicized:

An insurance company's core business depends on its members. It is important to know how many members the company has over **time**, whether the *number of members* is increasing or decreasing, and which **health plan** is doing well over time.

We don't have many dimensions attached to this fact based on the business process, but that's OK because it supports our reporting questions. We have two main dimensions: the date and health plan. Our metrics are the number of members that we have. Note that we don't have a member dimension because we are looking at an aggregation of the number of members over time, not individual members. It is important to provide this data on a time basis so that people can report historical information. Our final snapshot table can be created with the following script:

www.it-ebooks.info

```
-- Create Membership snapshot fact table
CREATE TABLE fact.Membership (
    DateKey INTEGER NOT NULL,
    HealthPlanKey INTEGER NOT NULL,
    MemberAmount INTEGER NOT NULL
)
GO
```

In addition, we need to add the foreign keys to link the fact table to our dimensions, using the following script:

```
-- Add foreign keys from Membership fact to dimensions
ALTER TABLE fact.Membership WITH CHECK
ADD CONSTRAINT FK_Membership_Date
FOREIGN KEY(DateKey) REFERENCES dim.Date (DateKey)
GO

ALTER TABLE fact.Membership WITH CHECK
ADD CONSTRAINT FK_Membership_HealthPlan
FOREIGN KEY(HealthPlanKey) REFERENCES dim.HealthPlan (HealthPlanKey)
GO
```

To load sample data into this table, use the following query:

```
-- Insert sample data into the Membership fact table
DECLARE @startdate DATE
DECLARE @enddate DATE
SET @startdate = '1/1/2011'
SET @enddate = '12/31/2011'
WHILE @startdate <= @enddate
BEGIN
        INSERT INTO fact.Membership
        SELECT CONVERT(CHAR(8), @startdate, 112) AS DateKey
              ,1 AS HPKey
              ,RAND() * 1000 AS MemberAmount
        INSERT INTO fact.Membership
        SELECT CONVERT(CHAR(8), @startdate, 112) AS DateKey
              ,2 AS HPKey
              ,RAND() * 1000 AS MemberAmount
        INSERT INTO fact.Membership
        SELECT CONVERT(CHAR(8), @startdate, 112) AS DateKey
              ,3 AS HPKey
              ,RAND() * 1000 AS MemberAmount

        SET @startdate = DATEADD(dd, 1, @startdate)
END
GO
```

This type of fact table can get very large, because it contains data for every combination of keys. Although, in this case, the key combination is just day and health plan, imaging adding a few more dimensions, and see how quickly that would grow. What we lose in storage, however, we gain in power. As an example, the following query quickly illustrates how many members plan 2 had over time. We can use this information to show a line chart or a dashboard that shows trends.

www.it-ebooks.info

```
select dd.DateValue, fm.MemberAmount
from fact.Membership fm
left join dim.Date dd on fm.DateKey = dd.DateKey
left join dim.HealthPlan dp on fm.HealthPlanKey = dp.HealthPlanKey
where dd.DateValue IN ('2011-09-01', '2011-09-02', '2011-09-03')
and fm.HealthPlanKey = 2
GO
```

This snapshot fact table is based on a daily grain, but showing snapshots on only a monthly grain instead is also common. This coarser grain can help the storage issue but should only be an alternative if the business process looks at the metrics on the monthly basis.

Analytical Querying

Recall that, from my outty perspective, the whole point of storing data is to get the information out, and we can write a variety of queries to get that information. Let's start with a combination of our dimensions and transaction fact from the previous section to provide some information to our users. Then, we'll look at how indexes can improve our query performance.

Queries

Our first reporting request is to deliver the number of claims that have had a payment denied over a certain length of time. For this query, we will use the `fact.ClaimPayment` table joined to some dimension tables:

```
select count(fcp.ClaimID) as DeniedClaimCount
from fact.ClaimPayment fcp
inner join dim.AdjudicationType da on fcp.AdjudicationTypeKey=da.AdjudicationTypeKey
inner join dim.Date dd  on fcp.DateKey=dd.DateKey
where da.AdjudicationType = 'DENIED'
and dd.MonthValue = 9
GO
```

This query highlights a few querying best practices. First of all, try not to filter by the surrogate key values. Using surrogate keys is tempting, because we know that the key for the `DENIED` adjudication type is always going to be 3. Doing so would also remove a join, because we could filter directly on the claim payment fact. However, this makes the query less readable in the future, not only to other report writers but also to us.

Additionally, this query shows the importance of creating descriptive column and table aliases. Creating column aliases that are descriptive will make it easier to understand the output of the query. Also, creating table aliases and using them to describe columns will reduce the possible confusion in the future if you add new columns or modify the query.

Because we were smart in creating our model, we can actually improve the performance of this query even further by using our `DeniedCount` column. Essentially, the work we need to do in the query was already done in the load process, so we end up with the following query:

```
select sum(fcp.DeniedCount) as DeniedClaimCount
from fact.ClaimPayment fcp
inner join dim.Date dd on fcp.DateKey=dd.DateKey
where dd.MonthValue = 9
GO
```


www.it-ebooks.info

Our next reporting query is to assist the operations department in determining how much we are paying out based on automatic adjudication per provider, as the company is always trying to improve that metric. We can use the same fact but modify our dimensions to create the following query:

```
select dp.OrganizationName, sum(fcp.AutoPayoutAmount) as AutoPayoutAmount
from fact.ClaimPayment fcp
inner join dim.Provider dp on fcp.ProviderKey=dp.ProviderKey
group by dp.OrganizationName
GO
```

Many of the principles from relational querying also apply to reporting queries. Among these are to make sure to use set-based logic where possible, avoid using `select *`, and restrict the query results whenever possible. If we take the previous query, we can improve it by adding a filter and focusing on the business need. Because the operations group wants to improve their automatic adjudication rate, it is looking at the value over time. Taking those two factors into account, our new query can be used to create a line chart of the automatic adjudication rate over time:

```
select dp.OrganizationName,
        dd.MonthValue,
        sum(fcp.AutoPayoutAmount)/sum(ClaimAmount)*100 as AutoRatio
from fact.ClaimPayment fcp
inner join dim.Provider dp on fcp.ProviderKey=dp.ProviderKey
inner join dim.Date dd on fcp.DateKey=dd.DateKey
where dd.DateValue between '01/01/2011' and '12/31/2011'
group by dp.OrganizationName, dd.MonthValue
GO
```

A request that I often receive is "I want everything." While I recommend digging into the business need to truly understand the reasoning behind the request, sometimes "everything" is exactly what the situation calls for. If so, we could provide a query similar to this:

```
select dd.DateValue, dm.InsuranceNumber, dat.AdjudicationType,
    dp.OrganizationName, ddiag.DiagnosisCode, dhcpc.ProcedureCode,
    SUM(fcp.ClaimAmount) as ClaimAmount,
    SUM(fcp.AutoPayoutAmount) as AutoPaymountAmount,
    SUM(fcp.ManualPayoutAmount) as ManualPayoutAmount,
    SUM(fcp.AutoAdjudicatedCount) as AutoAdjudicatedCount,
    SUM(fcp.ManualAdjudicatedCount) as ManualAdjudicatedCount,
    SUM(fcp.DeniedCount) as DeniedCount
from fact.ClaimPayment fcp
inner join dim.Date dd  on fcp.DateKey=dd.DateKey
inner join dim.Member dm  on fcp.MemberKey=dm.MemberKey
inner join dim.AdjudicationType dat on fcp.AdjudicationTypeKey=dat.AdjudicationTypeKey
inner join dim.Provider dp  on fcp.ProviderKey=dp.ProviderKey
inner join dim.Diagnosis ddiag  on fcp.DiagnosisKey=ddiag.DiagnosisKey
inner join dim.HCPCSProcedure dhcpc on fcp.ProcedureKey=dhcpc.ProcedureKey
group by dd.DateValue, dm.InsuranceNumber, dat.AdjudicationType,
 dp.OrganizationName, ddiag.DiagnosisCode, dhcpc.ProcedureCode
GO
```

The result of this query provides a great base for a drill-through report, where an end user can see a high-level summary of information and drill into the details. Additional fields from each dimension can be included without affecting the grain of the query at this time.

www.it-ebooks.info

Notice that, in each of the preceding queries, we start with the fact table and then add the appropriate dimensions. Using this approach focuses on the metrics and uses the entities as descriptors. If you every find that you are querying a dimension directly, revisit the business process, and see if you can create a new fact to alleviate this concern.

Indexing

Indexing is typically used to increase the query speed, so it's a perfect fit for reporting. Let's look at the following query based off of our healthcare model and work through some performance tuning through indexing to speed up the query:

```
SELECT ProcedureKey, SUM(ClaimAmount) As ClaimByProcedure
FROM fact.ClaimPayment
GROUP BY ProcedureKey
GO
```

With no index, this results in the following query plan:

```
 |--Stream Aggregate(GROUP BY:([ReportDesign].[fact].[ClaimPayment].[ProcedureKey])
DEFINE:([Expr1004]=SUM([ReportDesign].[fact].[ClaimPayment].[ClaimAmount])))

   |--Sort(ORDER BY:([ReportDesign].[fact].[ClaimPayment].[ProcedureKey] ASC))

      |--Table Scan(OBJECT:([ReportDesign].[fact].[ClaimPayment]))
```

The table scan is a sign that this query may take longer than we would wish. Let's start with a nonclustered index with our procedure key to see how that speeds up our performance. Create the index with this script:

```
CREATE NONCLUSTERED INDEX NonClusteredIndex ON fact.ClaimPayment
(
  ProcedureKey ASC
)
GO
```

By running the same query, we will see the same plan, so we were not able to make any performance gains. Next, let's try a new feature in SQL Server 2012 called *columnstore indexes*. Rather than store index data per row, they store data by column. They are typical indexes in that they don't change the structure of the data, but they do make the table read-only in SQL Server 2012 because of the complexity of maintaining them. For each of the columns referenced in a columnstore index, all of the data is grouped and stored together. A table may have only a single columnstore index, but because of the way that index is structured, every column referenced is basically valuable independently, unlike existing row indexes. Create a columnstore index using the following script:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX ColumnStoreIndex ON fact.ClaimPayment
(
    DateKey,
    MemberKey,
    AdjudicationTypeKey,
    ProviderKey,
    DiagnosisKey,
    ProcedureKey,
    ClaimID,
```


www.it-ebooks.info

```
    ClaimAmount,
    AutoPayoutAmount,
    ManualPayoutAmount,
    AutoAdjudicatedCount,
    ManualAdjudicatedCount,
    DeniedCount
)
GO
```

When you run the query again, you can see that the plan has changed quite drastically:

```
 |--Stream Aggregate(GROUP BY:([ReportDesign].[fact].[ClaimPayment].[ProcedureKey])
DEFINE:([Expr1004]=SUM([ReportDesign].[fact].[ClaimPayment].[ClaimAmount])))
   |--Sort(ORDER BY:([ReportDesign].[fact].[ClaimPayment].[ProcedureKey] ASC))
      |--Index Scan(OBJECT:([ReportDesign].[fact].[ClaimPayment].[ColumnStoreIndex]))
```

The index scan operator (rather than the table scan operator from the original query) shows that this query is working with a columnstore index. I won't cover columnstore indexes in any more detail, but for fact tables and some dimensions with large numbers of rows and low cardinality columns, you may be able to get order of magnitude improvements in performance by simply adding this one index. For more details, check SQL Server Books Online in the topic "columnstore indexes."

Summary Modeling for Aggregation Reporting

If you are interested in analytical reporting, a dimensional model will probably best suit your needs. On the other hand, in some scenarios, a full dimensional model is not necessary. Creating an aggregation table or set of tables with the desired information will work perfectly for you in these cases. This section covers the reasons why you would want to use summary tables over dimensional modeling, the pros and cons of using a summary table, and finally, how to create a summary table.

Methodologies do not exist for summary modeling, as the development is so specific to your exact need. You can create summary tables to show values over specific time frames, rollups of different children companies versus the entire parent company, or even every requested value for a customer. We will cover an example of something I have used successfully in the past.

Summary modeling is best used when the slicing and dicing that we normally associate with data warehousing is not necessary. The information provided from the table usually feeds a limited set of reports but is tied very tightly to the information needed. Some of the benefits of summary modeling follow:

- Increased speed due to the preaggregated values
- Ease of maintenance and understandability due to fewer joins

Before jumping into this solution, you'll want to consider some disadvantages as well:

- fixed data granularity limits the type of reports that can be created.
- Details of the data are not available for further analytical research.
- Historical tracking, such as what is provided with slowly changing dimensions, is not available.

www.it-ebooks.info

Initial Summary Table

Once you've decided that the benefits outweigh the disadvantages, you are ready to begin the process. You must go through the same steps as described in the "Requirements Gathering Process" section earlier in this chapter. You start this section once you reach step 4.

Let's use the same business process as described earlier: When an insurance company receives a claim for a member, the claim goes through an adjudication process. The claim can either be automatically adjudicated, manually adjudicated, or denied. As an insurance company, we need to know whether we adjudicated the claim, how much the physician requested based on the member's diagnosis, and how much we are willing to pay for that procedure.

In addition, we've been told that we want to display the number of claims over time for each adjudication type in a report. We could show this information using the prior model, but how would we do this in a summary model? We will walk through the steps now to produce the final model shown in Figure 14-4.

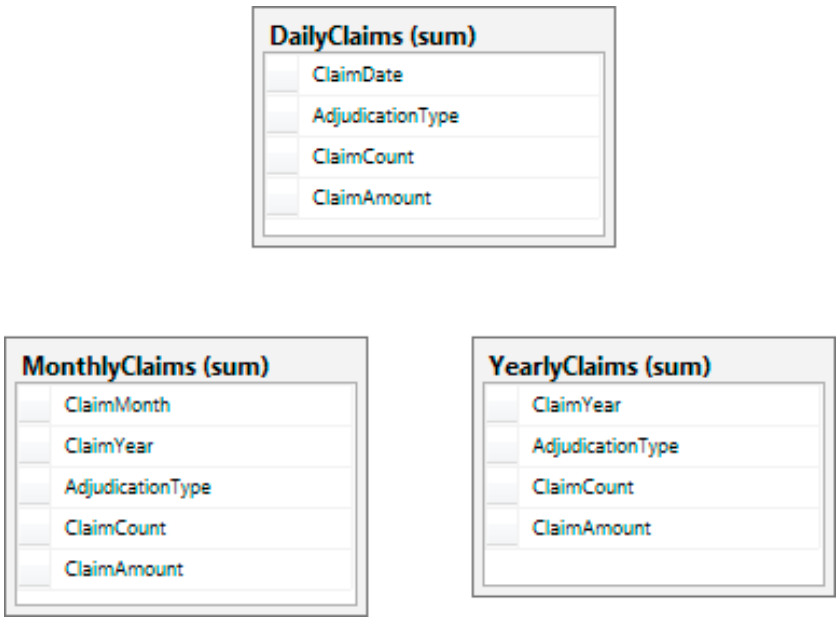


Figure 14-4. *Summary model for health care claim data*

Our first step is to determine the grain of our table. Using the business process description and report requirements, we see the grain includes: day, adjudication type, claim count, and claim amount. Unlike dimensional modeling, all codes and descriptive values are included in the aggregation table. Summary tables also do not use surrogate keys because there is no need to join tables together.

The code to create this table is shown here:

```
-- Create schema for summary tables
CREATE SCHEMA [sum]
GO
```


www.it-ebooks.info

```
-- Create Daily Claims table
CREATE TABLE [sum].DailyClaims (
    ClaimDate DATE NOT NULL,
    AdjudicationType VARCHAR(6) NOT NULL,
    ClaimCount INTEGER NOT NULL,
    ClaimAmount DECIMAL(10,2) NOT NULL
)
GO
```

Note that the when you load the data, you will need to preaggregate the values to match your grain. For example, your claims system probably records more than one claim per day, but you want to see them all together! You can use SQL statements with SUM functions or Integration Services packages with the aggregate task, but either way, your insert should be the complete set of values. For a sample set of data, run the following query:

```
-- Add sample data for summary tables
DECLARE @i INT
SET @i = 0

WHILE @i < 1000
BEGIN
INSERT INTO sum.DailyClaims
(
    ClaimDate, AdjudicationType, ClaimCount, ClaimAmount
)
SELECT
    CONVERT(CHAR(8), DATEADD(dd, RAND() * -100, getdate()), 112),
    CASE CEILING(3 * RAND())
    WHEN 1 THEN 'AUTO'
    WHEN 2 THEN 'MANUAL'
    ELSE 'DENIED'
    END,
    1,
    RAND() * 100000
SET @i = @i + 1
END
GO
```

The results that would be stored in the table could look similar to the following:

```
ClaimDate    AdjudicationType   ClaimCount   ClaimAmount
----------   ----------------   ----------   -----------
2011-09-23   AUTO               1            54103.93
2011-09-05   DENIED             1            30192.30
2011-08-26   MANUAL             1            9344.87
2011-09-06   DENIED             1            29994.54
2011-06-25   AUTO               1            52412.47
```


www.it-ebooks.info

Additional Summary Tables

We've only completed part of our model though. What if report writers want to look at the adjudication types on a monthly, rather than daily, basis? Or even on a yearly basis? To satisfy these requirements, we will create two more tables:

```
-- Create Monthly Claims table
CREATE TABLE [sum].MonthlyClaims (
    ClaimMonth INTEGER NOT NULL,
    ClaimYear INTEGER NOT NULL,
    AdjudicationType VARCHAR(6) NOT NULL,
    ClaimCount INTEGER NOT NULL,
    ClaimAmount DECIMAL(10,2) NOT NULL
)
GO

-- Create Yearly Claims table
CREATE TABLE [sum].YearlyClaims (
    ClaimYear INTEGER NOT NULL,
    AdjudicationType VARCHAR(6) NOT NULL,
    ClaimCount INTEGER NOT NULL,
    ClaimAmount DECIMAL(10,2) NOT NULL
)
GO
```

These tables can be loaded using the same method as before or by aggregating from the smaller-grained tables. Sample queries to load the month and year tables are shown here:

```
-- Add summarized data
insert into sum.MonthlyClaims
select MONTH(ClaimDate), YEAR(ClaimDate), AdjudicationType,
    SUM(ClaimCount), SUM(ClaimAmount)
from sum.DailyClaims
group by MONTH(ClaimDate), YEAR(ClaimDate), AdjudicationType
GO

insert into sum.YearlyClaims
select YEAR(ClaimDate), AdjudicationType, SUM(ClaimCount), SUM(ClaimAmount)
from sum.DailyClaims
group by YEAR(ClaimDate), AdjudicationType
GO
```

When creating a model such as this one, it is tempting to create one supertable that includes all of the date grains: daily, monthly, and yearly. A table that does this would have another column named `DateType` where that row would describe each grain. While this option is valid, I prefer to make querying and maintenance easier and keep these as separate tables.

Aggregation Querying

Because this is the section on reporting, we must talk about getting the information out of the tables we've designed! By design, the queries for summary tables are usually quite simple. We do have a few tricks up our sleeves to get the most out of our queries. Let's look at some queries using the different grain table we created and then look at the best way to index the tables to get the best performance.

www.it-ebooks.info

Queries

Let's begin with the original request we received from the business: a way to see the number of claims over time for different adjudication types. Depending on what time the users are interested in, they will use one of the following very simple queries:

```
SELECT ClaimDate, AdjudicationType, ClaimCount, ClaimAmount
FROM sum.DailyClaims
GO

SELECT ClaimMonth, ClaimYear, AdjudicationType, ClaimCount, ClaimAmount
FROM sum.MonthlyClaims
GO

SELECT ClaimYear, AdjudicationType, ClaimCount, ClaimAmount
FROM sum.YearlyClaims
GO
```

This will return the exact information requested by the end user. It is important to still drill into what the end result of the report should look like for end users. For example, if they are only interested in seeing the last three months of data at the day level, we can write that query as such:

```
SELECT ClaimDate, AdjudicationType, ClaimCount, ClaimAmount
FROM sum.DailyClaims
WHERE ClaimDate BETWEEN DATEADD(mm, -3, getdate()) and getdate()
ORDER BY ClaimDate
GO
```

The previous query filters the data returned so that a smaller set of information needs to be consumed and the query will perform more quickly. This best practice is similar to others we've discussed, and a lot of the other query best practices that we've talked about previously in this book are also applicable on summary tables.

Our next query request deals with the output of the information. We need to see the claim amounts for each of the adjudication types split into separate columns to be able to easily create a line chart. In this scenario, we will use the `MonthlyClaims` table and the SQL Server `PIVOT` function. The new query is shown here:

```
SELECT ClaimMonth, ClaimYear, [AUTO], [MANUAL], [DENIED]
FROM
(SELECT ClaimMonth, ClaimYear, AdjudicationType, ClaimAmount
  FROM sum.MonthlyClaims) AS Claims
PIVOT
(
SUM(ClaimAmount)
FOR AdjudicationType IN ([AUTO], [MANUAL], [DENIED])
) AS PivotedClaims
GO
```

This query works because we have a set list of values in the `AdjudicationType` field. Once we've successfully pivoted the information from rows to columns, the end user can use the results to create a report similar to the one shown in Figure 14-5.

www.it-ebooks.info

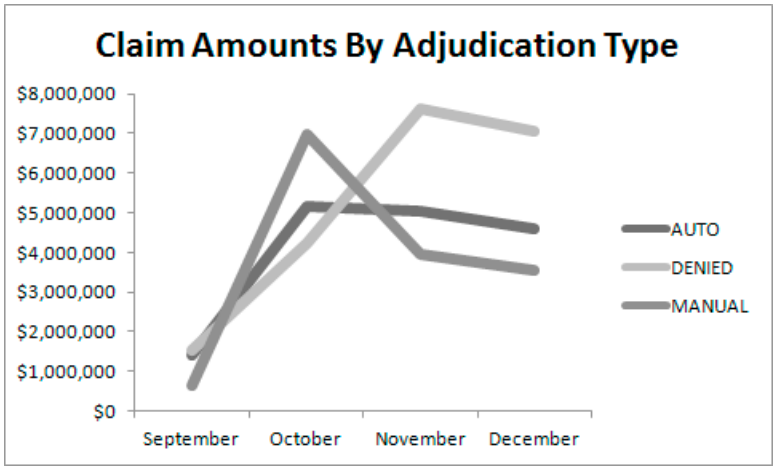


Figure 14-5. *Sample claim amount report*

Indexing

Indexing is particularly important within summary tables, as users will often be searching for just one row. In particular, a covering index is a perfect fit. Let's use a variation of our first simple query and build an index to support it. The query is

```
select SUM(ClaimCount) from sum.DailyClaims
where ClaimDate = '09/10/2011'
GO
```

With no index, we get a fairly poor plan with a table scan and stream aggregate, as shown here:

```
 |--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1006]=(0) THEN NULL ELSE [Expr1007] END))
    |--Stream Aggregate(DEFINE:([Expr1006]=Count(*), [Expr1007]=SUM([ReportDesign].[sum].
[DailyClaims].[ClaimCount])))
    |--Table Scan(OBJECT:([ReportDesign].[sum].[DailyClaims]), WHERE:([ReportDesign].[sum].
[DailyClaims].[ClaimDate]=CONVERT_IMPLICIT(date,[@1],0)))
```

Luckily for us, we know that a covering index that includes the column in the where clause and the column being selected will greatly improve the performance. Create the index using the following script:

```
CREATE NONCLUSTERED INDEX NonClusteredIndex ON sum.DailyClaims
(
    ClaimDate ASC,
    ClaimCount
)
GO
```


www.it-ebooks.info

We end up with the plan shown here, in which the index seek will definitely improve our performance:

```
 |--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1006]=(0) THEN NULL ELSE [Expr1007] END))

    |--Stream Aggregate(DEFINE:([Expr1006]=Count(*), [Expr1007]=SUM([ReportDesign].[sum].
[DailyClaims].[ClaimCount])))

    |--Index Seek(OBJECT:([ReportDesign].[sum].[DailyClaims].[NonClusteredIndex]),
SEEK:([ReportDesign].[sum].[DailyClaims].[ClaimDate]=CONVERT_IMPLICIT(date,[@1],0))
ORDERED FORWARD)
```

Make sure to always create the index based on the intended usage of the table. Creating the correct indexing scheme will make both you and your end users happier!

Summary

This chapter took a sharp turn off the path of relational modeling to delve into the details of modeling for reporting. We covered why you would want to model a reporting database differently than a transactional system and the different ways we can model for reporting. I definitely did not cover every potential scenario you may face, but I hope I gave you a good introduction to get started with some initial reporting structures.

Dimensional modeling for analytical reporting and summary modeling for aggregate reporting serve different needs and requirements. Be sure to use the one most appropriate for your environment. Dimensional modeling, the better-known method, has several well-known methodologies, lead by Ralph Kimball and Bill Inmon. This chapter used ideas from both men, while explaining clearly and concisely some general dimensional modeling concepts. Summary modeling is a great first step toward reporting and can be very powerful. Be sure to understand the pros and cons of using this type of solution.

Finally, we looked at querying the different types of models. When designing for reporting, the end user and final queries should always be in the forefront. Without people who query, you have no need for a reporting database!

www.it-ebooks.info

APPENDIX A

Scalar Datatype Reference

Choosing proper datatypes to match the domain chosen in logical modeling is an important task. One datatype might be more efficient than another of a similar type. For example, you can store integer data in an integer datatype, a numeric datatype, a floating point datatype, a character type, or even a binary column, but these datatypes certainly aren't alike in implementation or performance.

In this appendix, I'll introduce you to all the intrinsic datatypes that Microsoft provides and discuss the situations where they're best used. The following is a list of the datatypes I'll cover. I'll discuss when to use them and in some cases why not to use them.

- *Precise numeric data*: Stores data with no loss of precision due to storage.
  - `bit`: Stores either `1`, `0`, or `NULL`. Used for Boolean-type columns.
  - `tinyint`: Non-negative values between `0` and `255`.
  - `smallint`: Integers between `-32,768` and `32,767`.
  - `int`: Integers between `2,147,483,648` and `2,147,483,647` (-2^{31} to $2^{31} - 1$).
  - `bigint`: Integers between `9,223,372,036,854,775,808` and `9,223,372,036,854,775,807` (that is, -2^{63} to $2^{63} - 1$) .
  - `decimal`: Values between $-10^{38} + 1$ through $10^{38} - 1$.
  - `money`: Values from `-922,337,203,685,477.5808` through `922,337,203,685,477.5807`.
  - `smallmoney`: Values from `-214,748.3648` through `+ 214,748.3647`.
- *Approximate numeric data*: Stores approximations of numbers. Provides for a large range of values.
  - `float (N)`: Values in the range from `-1.79E + 308` through `1.79E + 308`.
  - `real`: Values in the range from `-3.40E + 38` through `3.40E + 38`. `real` is a synonym for a `float(24)` datatype.
- *Date and time*: Stores date values, including time of day.
  - `date`: Date-only values from January 1, 0001 to December 31, 9999 (3 bytes).
  - `time`: Time of day–only values to 100 nanoseconds (3–5 bytes). Note that the range of this type is from 0:00 to 23:59:59 and some fraction of a second based on the precision you select.

www.it-ebooks.info

    - `datetime2`: Despite the hideous name, this type will store dates from January 1, 0001 to December 31, 9999, to 100-nanosecond accuracy (6–8 bytes). The accuracy is based on the precision you select.
    - `datetimeoffset`: Same as `datetime2` but includes an offset from UTC time (8–10 bytes).
    - `smalldatetime`: Dates from January 1, 1900 through June 6, 2079, with accuracy to 1 minute (4 bytes). (Note: it is suggested to phase out usage of this type and use the more standards-oriented `datetime2`, though `smalldatetime` is not technically deprecated.)
    - `datetime`: Dates from January 1, 1753 to December 31, 9999, with accuracy to ~3 milliseconds (stored in increments of .000, .003, or .007 seconds) (8 bytes). (Note: it is suggested to phase out usage of this type and use the more standards-oriented `datetime2`, though `datetime` is not technically deprecated)
- *Character (or string) data*: Used to store textual data, such as names, descriptions, notes, and so on.
    - `char`: Fixed-length character data up to 8,000 characters long.
    - `varchar:` Variable-length character data up to 8,000 characters long.
    - `varchar(max)`: Large variable-length character data; maximum length of $2^{31} - 1$ (2,147,483,647) bytes, or 2 GB.
    - `text`: Large text values; maximum length of $2^{31} - 1$ (2,147,483,647) bytes, or 2 GB. (Note that this datatype is outdated and should be phased out in favor of the `varchar(max)` datatype.)
    - `nchar`, `nvarchar`, `ntext`: Unicode equivalents of `char`, `varchar`, and `text` (with the same deprecation warning for `ntext` as for `text`).
- *Binary data*: Data stored in bytes, rather than as human-readable values, for example, files or images.
    - `binary`: Fixed-length binary data up to 8,000 bytes long.
    - `varbinary`: Variable-length binary data up to 8,000 bytes long.
    - `varbinary(max)`: Large binary data; maximum length of $2^{31} - 1$ (2,147,483,647) bytes, or 2 GB.
    - `image`: Large binary data; maximum length of $2^{31} - 1$ (2,147,483,647) bytes, or 2 GB. (Note that this datatype is outdated and should be phased out for the `varbinary(max)` datatype.)
- *Other scalar datatypes*: Datatypes that don't fit into any other groups nicely, but are still interesting.
    - `rowversion` (or `timestamp`): Used for optimistic locking.
    - `uniqueidentifier`: Stores a globally unique identifier (GUID) value.
    - `cursor`: Datatype used to store a cursor reference in a variable. Cannot be used as a column in a table.

www.it-ebooks.info

    - `table`: Used to hold a reference to a local temporary table. Cannot be used as a column in a table.
    - `sql_variant`: Stores data of most any datatype.
- *Not simply scalar*: For completeness, I mention these types, but they are not covered in any detail. These types are `XML`, `hierarchyId`, and the spatial types (geometry and geography). `hierarchyId` is used in Chapter 8 when dealing with hierarchical data patterns.

Although we'll look at all these datatypes, this doesn't mean you'll have a need for all of them. Choosing a datatype needs to be a specific task to meet the needs of the client with the proper datatype. You could just store everything in unlimited-length character strings (this was how some systems worked in the old days), but this is clearly not optimal. From the list, you'll choose the best datatype, and if you cannot find one good enough, you can use the CLR and implement your own. The proper datatype choice is the first step in making sure the proper data is stored for a column.

■ **Note** I include information in each section about how the types are affected by using compression. This information refers to row-level compression only. For page-level compression information, see Chapter 10. Also note that compression is available only in the Enterprise Edition of SQL Server 2008 (and later).

Precise Numeric Data

You can store numerical data in many base datatypes, depending upon the actual need you are trying to fill. There are two different classes of numeric data: precise and approximate. The differences are important and must be well understood by any architect who's building a system that stores readings, measurements, or other numeric data.

Precise values have no error in the way they're stored, from integer to floating point values, because they have a fixed number of digits before and after the decimal point (or *radix*).

Approximate datatypes don't always store exactly what you expect them to store. However, they are useful for scientific and other applications where the range of values varies greatly.

The precise numeric values include the `bit`, `int`, `bigint`, `smallint`, `tinyint`, `decimal`, and `money datatypes` (`money` and `smallmoney`). I'll break these down again into three additional subsections: whole numbers and fractional numbers. This is done so we can isolate some of the discussion down to the values that allow fractional parts to be stored, because quite a few mathematical "quirks" need to be understood surrounding using those datatypes. I'll mention a few of these quirks, most importantly with the money datatypes. However, when you do any math with computers, you must be careful how rounding is achieved and how this affects your results.

Integer Values

Whole numbers are, for the most part, integers stored using base-2 values. You can do bitwise operations on them, though generally it's frowned upon in SQL. Math performed with integers is generally fast because the CPU can perform it directly using registers. I'll cover five integer sizes: `tinyint`, `smallint`, `int`, and `bigint`.

The biggest thing that gets a lot of people is math with integers. Intuitively, when you see an expression like:

```
SELECT 1/2;
```


www.it-ebooks.info

You will immediately expect that the answer is .5. However, this is not the case because integers don't work this way; integer math only return integer values. The next intuitive leap you will probably make is that .5 should round up to 1, right? Nope, even the following query:

```
SELECT CAST(.99999999 AS integer);
```

returns 0. Instead of rounding, integer math truncates values, because it performs math like you did back in elementary school. For example, consider the following equation, 305, divided by 100:

```
      3 R 5
100 ⟌ 305
    -300
      5
```

In a query, you get the whole number result using the division operator, and `to` get the remainder, you use the modulo operator (%). So you could execute the following query to get the division answer and the remainder:

```
SELECT 305 / 100, 305 % 100;
```

This returns 3 and 5. (The modulo operator is a very useful operator indeed.) If you want the result to be a non-whole number you need to cast at least one of the values to a datatype with a fractional element, like numeric, either by using cast, or a common method is to cast one of the factors to numeric, or by simply multiplying the first factor by 1.0:

```
SELECT CAST(305 AS numeric)/ 100, (305 * 1.0) / 100;
```

These mathematical expressions now both return 3.050000, which is the value that the user is most likely desiring to get, much like the person dividing 1 by 2 expects to get .5.

tinyint

Domain: Non-negative whole numbers from `0` through `255`.

Storage: 1 byte.

Discussion:

`tinyints` are used to store small non-negative integer values. When using a single byte for storage, if the values you'll be dealing with are guaranteed always to be in this range, a `tinyint` is perfect. A great use for this is for the primary key of a domain table that can be guaranteed to have only a couple values. The `tinyint` datatype is especially useful in a data warehouse to keep the surrogate keys small. However, you have to make sure that there will never be more than 256 values, so unless the need for performance is incredibly great (such as if the key will migrate to tables with billions and billions of rows), it's best to use a larger datatype.

Row Compression Effect:

No effect, because 1 byte is the minimum for the integer types other than `bit`.

smallint

Domain: Whole numbers from `-32,768` through `32,767` (or -2^{15} through $2^{15} - 1$).

Storage: 2 bytes.

www.it-ebooks.info

Discussion:

If you can be guaranteed to need values only in this range, the `smallint` can be a useful type. It requires 2 bytes of storage.

One use of a `smallint` that crops up from time to time is as a Boolean. This is because, in earlier versions of Visual Basic, `0` equaled `False` and `-1` equaled `True` (technically, VB would treat any nonzero value as `True`, but it used `-1` for its representation of `False`). Storing data in this manner is not only a tremendous waste of space—2 bytes versus potentially 1/8th of a byte for a bit, or even a single byte for a `char(1)`—`'Y'` or `'N'`. It's also confusing to all the other SQL Server programmers. ODBC and OLE DB drivers do this translation for you, but even if they didn't, it's worth the time to write a method or a function in VB to translate `True` to a value of `1`.

Row Compression Effect:

The value will be stored in the smallest number of bytes required to represent the value. For example, if the value is 10, it would fit in a single byte; then it would use 1 byte, and so forth, up to 2 bytes.

int

Domain: Whole numbers from `-2,147,483,648` to `2,147,483,647` (that is, -2^{31} to $2^{31} - 1$).

Storage: 4 bytes.

Discussion:

The integer datatype is frequently employed as a primary key for tables, because it's small (it requires 4 bytes of storage) and efficient to store and retrieve.

One downfall of the `int` datatype is that it doesn't include an unsigned version, which for a 32-bit version could store non-negative values from `0` to `4,294,967,296` (2^{32}). Because most primary key values start out at `1`, this would give you more than 2 billion extra values for a primary key value without having to involve negative numbers that can be confusing to the user. This might seem unnecessary, but systems that have billions of rows are becoming more and more common.

An application where the storage of the `int` column plays an important part is the storage of IP addresses as integers. An IP address is simply a 32-bit integer broken down into four octets. For example, if you had an IP address of 234.23.45.123, you would take $(234 * 2^3) + (23 * 2^2) + (45 * 2^1) + (123 * 2^0)$. This value fits nicely into an unsigned 32-bit integer, but not into a signed one. However, the 64-bit integer (`bigint`, which is covered next) in SQL Server covers the current IP address standard nicely but requires twice as much storage. Of course, `bigint` will fall down in the same manner when we get to IPv6 (the forthcoming Internet addressing protocol), because it uses a full 64-bit unsigned integer.

Row Compression Effect:

The value will be stored in the smallest number of bytes required to represent the value. For example, if the value is `10`, it would fit in a single byte; then it would use 1 byte, and so forth, up to 4 bytes.

www.it-ebooks.info

bigint

Domain: Whole numbers from `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807` (that is, -2^{63} to $2^{63} - 1$).

Storage: 8 bytes.

Discussion:

One of the common reasons to use the 64-bit datatype is as a primary key for tables where you'll have more than 2 billion rows, or if the situation directly dictates it, such as the IP address situation I previously discussed. Of course, there are some companies where a billion isn't really a very large number of things to store or count, so using a `bigint` will be commonplace to them. As usual, the important thing is to size your utilization of any type to the situation, not using too small or even too large of a type than is necessary.

Row Compression Effect:

The value will be stored in the smallest number of bytes required to represent the value. For example, if the value is `10`, it would fit in a single byte; then it would use 1 byte, and so forth, up to 8 bytes.

Decimal Values

The `decimal` datatype is precise, in that whatever value you store, you can always retrieve it from the table. However, when you must store fractional values in precise datatypes, you pay a performance and storage cost in the way they're stored and dealt with. The reason for this is that you have to perform math with the precise decimal values using SQL Server engine code. On the other hand, math with IEEE floating point values (the `float` and `real` datatypes) can use the floating point unit (FPU), which is part of the core processor in all modern computers. This isn't to say that the `decimal` type is slow, *per se*, but if you're dealing with data that doesn't require the perfect precision of the `decimal` type, use the `float` datatype. I'll discuss the `float` and `real` datatypes more in the "Approximate Numeric Data" section.

decimal (alias: numeric)

Domain: All numeric data (including fractional parts) between $-10^{38} + 1$ through $10^{38} - 1$.

Storage: Based on precision (the number of significant digits): 1–9 digits, 5 bytes; 10–19 digits, 9 bytes; 20–28 digits, 13 bytes; and 29–38 digits, 17 bytes.

Discussion:

The `decimal` datatype is a precise datatype because it's stored in a manner that's like character data (as if the data had only 12 characters, 0 to 9 and the minus and decimal point symbols). The way it's stored prevents the kind of imprecision you'll see with the `float` and `real` datatypes a bit later. However, `decimal` does incur an additional cost in getting and doing math on the values, because there's no hardware to do the mathematics.

To specify a decimal number, you need to define the precision and the scale:

- *Precision* is the total number of significant digits in the number. For example, 10 would need a precision of 2, and 43.00000004 would need a precision of 10. The precision may be as small as 1 or as large as 38.

www.it-ebooks.info

- *Scale* is the possible number of significant digits to the right of the decimal point. Reusing the previous example, 10 would require a scale of 0, and 43.00000004 would need 8.

Numeric datatypes are bound by this precision and scale to define how large the data is. For example, take the following declaration of a numeric variable:

```
DECLARE @testvar decimal(3,1)
```

This allows you to enter any numeric values greater than `-99.94` and less than `99.94`. Entering `99.949999` works, but entering `99.95` doesn't, because it's rounded up to `100.0`, which can't be displayed by `decimal(3,1)`. Take the following, for example:

```
SELECT @testvar = -10.155555555;
SELECT @testvar;
```

This returns the following result:

```
-------------------
-10.2
```

This rounding behavior is both a blessing and a curse. You must be careful when butting up to the edge of the datatype's allowable values. There is a setting—`SET NUMERIC_ROUNDABORT ON`— that causes an error to be generated when a loss of precision would occur from an implicit data conversion. That's kind of like what happens when you try to put too many characters into a character value.

Take the following code:

```
SET NUMERIC_ROUNDABORT ON;

DECLARE @testvar decimal(3,1);

SELECT @testvar = -10.155555555;

SET NUMERIC_ROUNDABORT OFF;--this setting persists for a connection
```

This causes the following error:

```
Msg 8115, Level 16, State 7, Line 3
Arithmetic overflow error converting numeric to data type numeric.
```

`SET NUMERIC_ROUNDABORT` can be quite dangerous to use and might throw off applications using SQL Server if set to `ON`. However, if you need guaranteed prevention of implicit round-off due to system constraints, it's there.

As far as usage is concerned, you should generally use the `decimal` datatype as sparingly as possible, and I don't mean this negatively. There's nothing wrong with the type at all, but it does take that little bit more processing than integers or real data, and hence there's a performance hit. You should use it when you have specific values that you want to store where you can't accept any loss of precision. Again, I'll deal with the topic of loss of precision in more detail in the section "Approximate Numeric Data." The `decimal` type is commonly used as a replacement for the `money` *type,* because it has certain round-off issues that `decimal` does not.

www.it-ebooks.info

Row Compression Effect:

The value will be stored in the smallest number of bytes that are necessary to provide the precision necessary, plus 2 bytes overhead per row. For example, if you are storing the value of `2` in a `numeric(28,2)` column, it needn't use all the possible space; it can use the space of a `numeric(3,2)`, plus the 2 bytes overhead.

Money Types

There are two intrinsic datatypes that are for storing monetary values. Both are based on integer types, with a fixed four decimal places. These types are as follows:

- `money`
  - *Domain*: `-922,337,203,685,477.5808` to `922,337,203,685,477.5807`
  - *Storage*: 8 bytes
- `smallmoney`
  - *Domain*: `-214,748.3648` to `214,748.3647`
  - *Storage*: 4 bytes

The money datatypes are generally considered a poor choice of datatype, even for storing monetary values, because they have a few inconsistencies that can cause a good deal of confusion. First, you can specify units, such as $ or £, but the units are of no real value. For example:

```
CREATE TABLE dbo.testMoney
(
  moneyValue money
);
go

INSERT INTO dbo.testMoney
VALUES ($100);
INSERT INTO dbo.testMoney
VALUES (100);
INSERT INTO dbo.testMoney
VALUES (£100);
GO
SELECT * FROM dbo.testMoney;
```

The query at the end of this code example returns the following results (each having the exact same value.):

```
moneyValue
----------
100.00
100.00
100.00
```

The second problem is that the money datatypes have well-known rounding issues with math. I mentioned that these types are based on integers (the range for `smallmoney` is `-214,748.3648` to `214,748.3647`, and the

www.it-ebooks.info

range for an integer is 2,147,483,648 to 2,147,483,647). Unfortunately, as I will demonstrate, intermediate results are stored in the same types, causing unexpected rounding errors. For example:

```
DECLARE @money1 money = 1.00,
        @money2 money = 800.00;

SELECT CAST(@money1/@money2 AS money);
```

This returns the following result:

```
---------------------
0.0012
```

However, try the following code:

```
DECLARE @decimal1 decimal(19,4) = 1.00,
      , @decimal2 decimal(19,4) = 800.00;

SELECT CAST(@decimal1/@decimal2 AS decimal(19,4));
```

It returns the following result:

```
----------------
0.0013
```

Why? Because money uses only four decimal places for intermediate results, where decimal uses a much larger precision:

```
SELECT @money1/@money2;
SELECT @decimal1/@decimal2;
```

This code returns the following results:

```
---------------------
0.0012

---------------------------------------
0.0012500000000000000
```

That's why there are round-off issues. And if you turned SET NUMERIC_ROUNDABORT ON, the decimal example would fail, telling you that you were losing precision, whereas there is no way to stop the roundoff from occurring with the money types. The common consensus among database architects is to avoid the money datatype and use a numeric type instead, because of the following reasons:

- Numeric types give the answers to math problems in the natural manner that's expected.
- Numeric types have no built-in units to confuse matters.

Even in the previous version of SQL Server, the following statement was included in the monetary data section: "If a greater number of decimal places are required, use the decimal datatype instead." Using a decimal type instead gives you the precision needed. To replicate the range for money, use DECIMAL(19,4), or for smallmoney, use DECIMAL(10,4). However, you needn't use such large values if you don't need them. If you happen to be calculating the national debt or my yearly gadget allowance, you might need to use a larger value.

www.it-ebooks.info

Row Compression Effect:

The money types are simply integer types with their decimal places shifted. As such, they are compressed in the same manner that integer types would be. However, since the values would be larger than they appear (because of the value `10` being stored as 10.000, or 10000 in the physical storage), the compression would be less than for an integer of the same magnitude.

Approximate Numeric Data

Approximate numeric values contain a decimal point and are stored in a format that's fast to manipulate. They are called *floating point* because they have a fixed number of significant digits, but the placement of the decimal point "floats," allowing for really small numbers or really large numbers. Approximate numeric values have some important advantages, as you'll see later in this appendix.

Approximate is such a negative term, but it's technically the proper term. It refers to the `real` and `float` datatypes, which are IEEE 75454 standard single- and double-precision floating point values. The number is stored as a 32-bit or 64-bit value, with four parts:

- *Sign*: Determines whether this is a positive or negative value.
- *Exponent*: The exponent in base-2 of the mantissa.
- *Mantissa*: Stores the actual number that's multiplied by the exponent (also known as the *coefficient* or *significand*).
- *Bias*: Determines whether the exponent is positive or negative.

A complete description of how these datatypes are formed is beyond the scope of this book but may be obtained from the IEEE body at `www.ieee.org`.

- `float [ (N) ]`
  - *Domain*: `-1.79E + 308` through `1.79E + 308`. The `float` datatype allows you to specify a certain number of bits to use in the mantissa, from 1 to 53. You specify this number of bits with the value in N. The default is 53.
  - *Storage*: See Table A-1

Table A-1. *Floating Point Precision and Storage Requirements*

N (Number of Mantissa Bits for Float)	Precision	Storage Size
1–24	7	4 bytes
25–53	15	8 bytes

- `real`
  - `real` is a synonym for `float(24)`.

At this point, SQL Server rounds all values of N up to either `24` or `53`. This is the reason that the storage and precision are the same for each of the values.

Discussion:

Using these datatypes, you can represent most values from `-1.79E + 308` to `1.79E + 308` with a maximum of 15 significant digits. This isn't as many significant digits as the

www.it-ebooks.info

numeric datatypes can deal with, but the range is enormous and is plenty for almost any scientific application (The distance from the Earth to the Andromeda galaxy is *only* 2.5 * power(10,19) kilometers!), . These datatypes have a much larger range of values than any other datatype. This is because the decimal point isn't fixed in the representation. In exact numeric types, you always have a pattern such as `NNNNNNN.DDDD` for numbers. You can't store more digits than this to the left or right of the decimal point. However, with `float` values, you can have values that fit the following patterns (and much larger):

- `0.DDDDDDDDDDDDDDD`
- `NNNNN.DDDDDDDDD`
- `0.OOOOOOOOOOOOOOOOOOOOOOOOOOOODDDDDDDDDDDDDDD`
- `NNNNNNNNNNNNNNNOOOOOOOOOOOOOOOOO`

So, you have the ability to store tiny numbers, or large ones. This is important for scientific applications where you need to store and do math on an extreme range of values. The `float` datatypes are well suited for this usage.

Row Compression Effect:

The least significant bytes with all zeros are not stored. This is applicable mostly to non-fractional values in the mantissa.

Date and Time Data

Back in SQL Server 2005, the choice of datatype for storing date and time data was pretty simple. We had two datatypes for working with date and time values: `datetime` and `smalldatetime`. Both had a time element and a date element, and you could not separate them. Not having a simple date or time datatype was be a real bother at times because a rather large percentage of data only really needs the date portion.

In SQL Server 2008, a set of new datatypes were added that changed all of that. These were date, time, `datetime2`, and `datetimeoffset`. These new datatypes represent a leap of functionality in addition to the original `datetime` and `smalldatetime`. For the most part, the new types cover the range of date and time values that could be represented in the original types, though only the `smalldatetime` type can easily represent a point in time to the minute, rather than second. As of the 2012 release of SQL Server, the `datetime` and `smalldatetime` are suggested to be phased out for future use, and as such you should move work to `datetime2` or one of the other date/time datatypes.

date

Domain: Date-only values from January 1, 0001 to December 31, 9999.

Storage: 3-byte integer, storing the offset from January 1, 0001.

Accuracy: One day.

Discussion:

Of all the features added back in SQL Server in 2008, this one datatype was worth the price of the upgrade (especially since I don't have to whip out my wallet and pay for it). The problem of how to store date values without time had plagued T-SQL programmers since the beginning of time (aka version 1.0).

www.it-ebooks.info

With this type, you will be able to avoid the tricks you have needed to go through to ensure that date types had no time in order to store just a date value. In past versions of the book, I advocated the creation of your own pseudodate values stored in an integer value. That worked, but you could not use the built-in date functions without doing some "tricks."

Row Compression Effect:

Technically you get the same compression as for any integer value, but dates in the "normal" range of dates require 3 bytes, meaning no compression is realized.

time [*(precision)*]

Domain: Time of day only (note this is not a quantity of time, but a point in time on the clock).

Storage: 3–5 bytes, depending on precision.

Accuracy: To 100 nanoseconds, depending on how it is declared. `time(0)` is accurate to 1 second, `time(1)` to .1 seconds, up to `time(7)` as .0000001. The default is 100-nanosecond accuracy.

Discussion:

The `time` type is handy to have but generally less useful. Initially it will seem like a good idea to store a point in time, but in that case you will have to make sure that both times are for the same day. Rather, for the most part when you want to store a time value, it is a point in time, and you need one of the date + time types.

The time value can be useful for storing a time for a recurring activity, for example, where the time is for multiple days rather than a single point in time.

Row Compression Effect:

Technically you get the same compression as for any integer value, but time values generally use most of the bytes of the integer storage, so very little compression should be expected for time values.

datetime2 [(precision)]

Domain: Dates from January 1, 0001 to December 31, 9999, with a time component.

Storage: Between 6 and 8 bytes. The first 4 bytes are used to store the date, and the others an offset from midnight, depending on the accuracy.

Accuracy: To 100 nanoseconds, depending on how it is declared. `datetime(0)` is accurate to 1 second, `datetime(1)` to .1 seconds, up to `datetime(7)` as .0000001. The default is 100-nanosecond accuracy.

Discussion:

This is a much better datatype than `datetime`. Technically, you get far better time support without being limited by the .003 accuracy issues that `datetime` is. The only downside I currently see is support for the type in your API.

www.it-ebooks.info

What I see as the immediate benefit of this type is to fix the amount of accuracy that your users actually desire. Most of the time a user doesn't desire fractional seconds, unless the purpose of the type is something scientific or technical. With `datetime2`, you can choose 1-second accuracy. Also, you can store .999 seconds, unlike `datetime`, which would round .999 up to 1, whereas .998 would round down to .997.

Row Compression Effect:

For the date portion of the type, dates before 2079 can save 1 byte of the 4 bytes for the date. Little compression should be expected for the time portion.

datetimeoffset [(precision)]

The `datetimeoffset` is the same as `datetime2`, but it includes an offset from UTC time (8–10 bytes).

Domain: Dates from January 1, 0001 to December 31, 9999, with a time component. Includes the offset from the UTC time, in a format of [+|-] hh:mm. (Note that this is not time zone/daylight saving time aware. It simply stores the offset at the time of storage.)

Storage: Between 8 and 10 bytes. The first 4 bytes are used to store the date, and just like `datetime2`, 2–4 will be used for the time, depending on the accuracy. The UTC offset is stored in the additional 2 bytes.

Accuracy: To 100 nanoseconds, depending on how it is declared. `datetimeoffset(0)` is accurate to 1 second, `datetimeoffset (1)` to .1 seconds, up to `datetimeoffset (7)` as .0000001. The default is 100-nanosecond accuracy.

Discussion:

The offset seems quite useful but in many ways is more cumbersome than using two date columns, one for UTC and one for local (though this will save a bit of space). Its true value is that it defines an exact point in time better than the regular `datetime` type, since there is no ambiguity as to where the time was stored.

A useful operation is to translate the date from its local offset to UTC, like this:

```
DECLARE @LocalTime DateTimeOffset;

SET @LocalTime = SYSDATETIMEOFFSET();
SELECT @LocalTime;

SELECT SWITCHOFFSET(@LocalTime, ' + 00:00') AS UTCTime;
```

The true downside is that it stores an offset, not the time zone, so daylight saving time will still need to be handled manually.

Row Compression Effect:

For the date portion of the type, dates before 2079 can save 1 byte of the 4 bytes for the date. Little compression should be expected for the time portion.

smalldatetime

Domain: Date and time data values between January 1, 1900 and June 6, 2079.

Storage: 4 bytes (two 2-byte integers: one for the day offset from January 1, 1900, the other for the number of minutes past midnight).

www.it-ebooks.info

Accuracy: One minute.

Discussion:

The `smalldatetime` datatype is accurate to 1 minute. It requires 4 bytes of storage. `smalldatetime` values are the best choice when you need to store the date and time of some event where accuracy of a minute isn't a problem.

`smalldatetime` is suggested to be phased out of designs and replaced with `datetime2`, though it is very pervasive and will probably be around for several versions of SQL Server yet to come. Unlike `datetime`, there is not a direct replacement in terms of accuracy, as minimum `datetime2` accuracy is to the second.

Row Compression Effect:

When there is no time stored, 2 bytes can be saved, and times less than 4 a.m. can save 1 byte.

datetime

Domain: Date and time data values between January 1, 1753 and December 31, 9999.

Storage: 8 bytes (two 4-byte integers: one for the day offset from January 1, 1753, and the other for the number of 3.33-millisecond periods past midnight).

Accuracy: 3.33 milliseconds.

Discussion:

Using 8 bytes, `datetime` is a bit heavy on memory, but the biggest issue is regarding the precision. The biggest issue is that it is accurate to .003 seconds, leading to interesting roundoff issues. For example, very often a person will write an expression such as:

```
WHERE datetimeValue < = '20110919 23:59:59.999'
```

This is used to avoid getting any values for the next day of '20110920 00:00:00.000'. However, because of the precision, the expression

```
SELECT CAST('20110919 23:59:59.999' AS datetime);
```

Will return: 2011-09-20 00:00:00.000. Instead, you will need to use

```
SELECT CAST('20110919 23:59:59.997' AS datetime);
```

To get the max `datetime` value that is less than '20110920 00:00:00.000'

`datetime` is suggested to be phased out of designs and replaced with `datetime2`, though it is very pervasive and will probably be around for several versions of SQL Server yet to come.

Row Compression Effect:

For the date portion of the type, dates before 2079 can save 1 byte. For the time portion, 4 bytes are saved when there is no time saved, and it uses the first 2 bytes after the first 2 minutes and reaches the fourth byte after 4 a.m. After 4 a.m., compression can generally save 1 byte.

www.it-ebooks.info

Discussion on All Date Types

Date types are often some of the most troublesome types for people to deal with. In this section, I'll lightly address the following problem areas:

- Date functions
- Date ranges
- Representing dates in text formats

Date Functions

With the creation of new `date` and `time` (and `datetime`) types back in SQL Server 2008, there needed to be more functions to work with. Microsoft has added functions that return and modify dates and time with more precision than `GETDATE` or `GETUTCDATE`:

- `SYSDATETIME`: Returns system time to the nearest fraction of a second, with seven places of scale in the return value
- `SYSDATETIMEOFFSET`: Same as `SYSDATETIME` but returns the offset of the server in the return value
- `SYSUTCDATETIME`: Same as `SYSDATETIME` but returns the UTC date time rather than local time
- `SWITCHOFFSET`: Changes the offset for a `datetimeoffset` value
- `TODATETIMEOFFSET`: Converts a local date and time value to a given time zone

There have been a few changes to the basic date functions as well:

- `DATENAME`: Includes values for microsecond, nanosecond, and `TZoffset`. These will work differently for the different types. For example, `TZoffset` will work with `datetime2` and `datetimeoffset`, but not the other types.
- `DATEPART`: Includes microsecond, nanosecond, `TZoffset`, and `ISO_WEEK`. `ISO_WEEK` is a feature that has been long desired by programmers who need to know the week of the year, rather than the nonstandard week value that SQL Server has previously provided.
- `DATEADD`: Supports micro and nanoseconds.

With all the date functions, you really have to be careful that you are cognizant of what you are requesting. For an example, how old is this person? Say you know that a person was born on December 31, 2008, and on January 3, 2009, you want to know their age. Common sense says to look for a function to take the difference between two dates. You are in luck—there is a `DATEDIFF` function. Executing the following:

```
DECLARE @time1 date = '20111231',
        @time2 date = '20120102';
SELECT DATEDIFF(yy,@time1,@time2);
```

You see that the person is 0 years old right? Wrong! It returns 1.Ok, so if that is true, then the following should probably return 2, right?

```
DECLARE @time1 date = '20110101',
        @time2 date = '20121231';
SELECT DATEDIFF(yy,@time1,@time2);
```


www.it-ebooks.info

That also returns 1. So, no matter whether the date values are 1 day apart or 730, you get the same result? Yes, because the DATEDIFF function is fairly dumb in that it is taking the difference of the year value, not the difference in years. Then, to find out the age, you will need to use several functions.

So, what is the answer? We could do something fancier, likely by coming up with an algorithm based on the number of days, or months, or shifting dates to some common value, but that is way more like work than the really straightforward answer. Build a table of dates, commonly called a *calendar* (see Chapter 12).

Date Ranges

This topic will probably seem really elementary, but the fact is that one of the largest blunders in the database implementation world is working with ranges of dates. The problem is that when you want to do inclusive ranges, you have always needed to consider the time in the equation. For example, the following criteria:

```
WHERE pointInTimeValue BETWEEN '2012-01-01' AND '2012-12-31'
```

means something different based on whether the values stored in pointInTimeValue have, or do not have, a time part stored. With the introduction of the date type, the need to worry about this issue of date ranges may eventually become a thing of the past, but it is still an issue that you'll always need to worry about whether you need to worry about it.

The problem is that any value with the same date as the end value plus a time (such as '2012-12-31 12:00:00') does not fit within the preceding selecting criteria. Because you will miss all the activity that occurred on December 31 that wasn't at midnight.

There are two ways to deal with this. Either code your WHERE clause like this:

```
WHERE pointInTimeValue > = '2012-01-01' AND pointInTimeValue < '2013-01-01'
```

or use a calculated column to translate point-in-time values in your tables to date-only values (like dateValue as cast(pointInTimeValue as date)). Many times the date value will come in handy for grouping activity by day as well. Having done that, a value such as '2012-12-31 12:00:00' will be truncated to '2012-12-31 00:00:00', and a row containing that value will be picked up by selection criteria such as this:

```
WHERE pointInTimeValue BETWEEN '2012-01-01' AND '2012-12-31'
```

A common solution that I don't generally suggest is to use a between range like this:

```
WHERE pointInTimeValue BETWEEN '2012-01-01' AND '2012-12-31 23:59:59.9999999'
```

The idea is that if the second value is less than the next day, values for the next day won't be returned. The major problem with this solution has to do with the conversion of 23:59:59.9999999 to one of the various date datatypes. Each of the types will round up, so you must match the number of fractional parts to the precision of the type. For datetime2(3), you would need 23:59:59.999. If the pointInTimeValue column was of type: datetime, you would need use this:

```
WHERE pointInTimeValue BETWEEN '2009-01-01' AND '2009-12-31 23:59:59.997'
```

However, for a smalldatetime value, it would need to be this:

```
WHERE pointInTimeValue BETWEEN '2009-01-01' AND '2009-12-31 23:59'
```

and so on, for all of the different date types, which gets complicated by the new types where you can specify precision. I strongly suggest you avoid trying to use a maximum date value like this unless you are tremendously careful with the types of data and how their values round off.

For more information about how date and time data work with one another and converting from one type to another, read the topic "Using Date and Time Data" in Books Online.

www.it-ebooks.info

Representing Dates in Text Formats

When working with date values in text, using a standard format is always best. There are many different formats used around the world for dates, most confusingly `MMDDYYYY` and `DDMMYYYY` (is `01022004` or `02012004` the same day, or a different day?). Although SQL Server uses the locale information on your server to decide how to interpret your date input, using one of the following formats ensures that SQL Server doesn't mistake the input regardless of where the value is entered.

Generally speaking, it is best to stick with one of the standard date formats that are recognized regardless of where the user is. This prevents any issues when sharing data with international clients, or even with sharing it with others on the Web when looking for help.

There are several standards formats that will work:

- ANSI SQL Standard
  - *No time zone offset*: `'YYYY-MM-DD HH:MM:SS'`
  - *With time zone*: `'YYYY-MM-DD HH:MM:SS –OH:OM'` (Z, for Zulu, can be used to indicate the time zone is the base time of 00:00 offset, otherwise known as GMT [Greenwich Mean Time] or the most standard/modern name is UTC [Coordinated Universal Time].)
- ISO 8601
  - *Unseparated*: `'YYYYMMDD'`
  - *Numeric*: `'YYYY-MM-DD'`
  - *Time*: `'HH:MM:SS.sssssss'` (`SS` and `.sssssss` are optional)
  - *Date and time*: `'YYYY-MM-DDTHH:MM:SS.sssssss'`
  - *Date and time with offset*: `'YYYY-MM-DDTHH:MM:SS.sssssss -OH:OM'`
- ODBC
  - *Date*: `{d 'YYYY-MM-DD'}`
  - *Time*: `{t 'HH:MM:SS'}`
  - *Date and time*: `{ts 'YYYY-MM-DD HH:MM:SS'}`

Using the ANSI SQL Standard or the ISO 8601 formats is generally considered the best practice for specifying date values. It will definitely feel odd when you first begin typing `'2008-08-09'` for a date value, but once you get used to it, it will feel natural.

The following are some examples using the ANSI and ISO formats:

```
SELECT CAST('2013-01-01' as date) AS dateOnly;
SELECT CAST('2013-01-01 14:23:00.003' AS datetime) as withTime;
```

You might also see values that are close to this format, such as the following:

```
SELECT CAST ('20130101' as date) AS dateOnly;
SELECT CAST('2013-01-01 T14:23:00.120' AS datetime) AS withTime;
```

For more information, check SQL Server 2012 Books Online under "Date and Time Format" Related to dates, a new function FORMAT has been added to SQL Server 2012 that will help you output dates in any format you need to. As a very brief example, consider the following code snippet:

www.it-ebooks.info

```
DECLARE @DateValue datetime2(3) = '2012-05-21 15:45:01.456'
SELECT  @DateValue as Unformatted,
        FORMAT(@DateValue,'yyyyMMdd') as IsoUnseperated,
        FORMAT(@DateValue,'yyyy-MM-ddThh:mm:ss') as IsoDateTime,
        FORMAT(@DateValue,'D','en-US' ) as USRegional,
        FORMAT(@DateValue,'D','en-GB' ) as GBRegional,
        FORMAT(@DateValue,'D','fr-fr' ) as FRRegional;
```

This returns the following:

```
Unformatted                 IsoUnseperated      IsoDateTime
--------------------------  ------------------  -------------------
2012-05-21 15:45:01.456     20120521            2012-05-21T03:45:01

USRegional                  GBRegional          FRRegional
--------------------------  ------------------  -------------------
Monday, May 21, 2012        21 May 2012         lundi 21 mai 2012
```

The unformatted version is simply how it appears in SSMS using my settings. The `IsoUnseperated` value was built using a format mask of `yyyyMMdd`, and the `IsoDateTime` using a bit more interesting mask, each of which should be fairly obvious, but check the `FORMAT` topic in books online for a full rundown of features. The last two examples format the date in the manner of a given region, each of which could come in very handy building regionalized reports. Note that the Great Britain version doesn't list the day of the week, whereas the United States and France do. `FORMAT` does more than just date data, but this is where we have generally felt the most pain with data through the years. so I mentioned it here.

There is another handy function `PARSE` that will let you take a value in a given regional version and parse information out of a formatted string. I won't demonstrate `PARSE`, but rather wanted to make you aware of more tools to work with date data here in this date data section of the appendix.

Character Strings

Most data that's stored in SQL Server uses character datatypes. In fact, usually far too much data is stored in character datatypes. Frequently, character columns are used to hold noncharacter data, such as numbers and dates. Although this might not be technically wrong, it isn't ideal. For starters, storing a number with eight digits in a character string requires at least 8 bytes, but as an integer it requires 4 bytes. Searching on integers is far easier because 1 always precedes 2, whereas 11 comes before 2 in character strings. Additionally, integers are stored in a format that can be manipulated using intrinsic processor functions, as opposed to having SQL Server–specific functions deal with the data.

char[(*length*)]

Domain: ASCII characters, up to 8,000 characters long.

Storage: 1 byte * length.

Discussion:

The `char` datatype is used for fixed-length character data. Every value will be stored with the same number of characters, up to a maximum of 8,000 bytes. Storage is exactly the number of bytes as per the column definition, regardless of actual data stored; any

www.it-ebooks.info

remaining space to the right of the last character of the data is padded with spaces. The default size if not specified is 1 (it is best practice to include the size).

You can see the possible characters by executing the following query:

```
SELECT number, CHAR(number)
FROM   Tools.Number
WHERE  number >=0 AND number <= 255;
```

■ **Tip** The `numbers` table is a common table that every database should have. It's a table of integers that can be used for many utility purposes. In Chapter 12, I present a numbers table that you can use for this query.

The maximum limit for a `char` is 8,000 bytes, but if you ever get within a mile of this limit for a fixed-width character value, you're likely making a big design mistake because it's extremely rare to have massive character strings of exactly the same length. You should employ the `char` datatype only in cases where you're guaranteed to have exactly the same number of characters in every row.

The `char` datatype is most often used for codes and identifiers, such as customer numbers or invoice numbers where the number includes alpha characters as well as integer data. An example is a vehicle identification number (VIN), which is stamped on most every vehicle produced around the world. Note that this is a composite attribute, because you can determine many things about the automobile from its VIN.

Another example where a `char` column is usually found is in Social Security numbers (SSNs), which always have nine characters and two dashes embedded.

Row Compression Effect:

Instead of storing the padding characters, it removes them for storage and adds them back whenever the data is actually used.

■ **Note** The setting `ANSI_PADDING` determines exactly how padding is handled. If this setting is `ON`, the table is as I've described; if not, data will be stored as I'll discuss in the "varchar(length)" section. It's best practice to leave this ANSI setting `ON`.

---

varchar[(*length*)]

Domain: ASCII characters, up to 8,000 characters long.

Storage: 1 byte * length + 2 bytes (for overhead).

Discussion:

For the `varchar` datatype, you choose the maximum length of the data you want to store, up to 8,000 bytes. The `varchar` datatype is far more useful than `char`, because the data doesn't have to be of the same length and SQL Server doesn't pad out excess memory with spaces. There's some reasonably minor overhead in storing variable-

www.it-ebooks.info

length data. First, it costs an additional 2 bytes per column. Second, it's a bit more difficult to get to the data, because it isn't always in the same location of the physical record. The default size if not specified is 1 (it is best practice to include the size).

Use the `varchar` datatype when your character data varies in length. The good thing about `varchar` columns is that, no matter how long you make the maximum, the space used by the column is based on the actual size of the characters being stored plus the few extra bytes that specify how long the data is.

You'll generally want to choose a maximum limit for your datatype that's a reasonable value, large enough to handle most situations, but not too large as to be impractical to deal with in your applications and reports. For example, take people's first names. These obviously require the `varchar` type, but how long should you allow the data to be? First names tend to be a maximum of 15 characters long, though you might want to specify 20 or 30 characters for the unlikely exception.

The most prevalent storage type for non-key values that you'll use is `varchar` data, because, generally speaking, the size of the data is one of the most important factors in performance tuning. The smaller the amount of data, the less has to be read and written. This means less disk access, which is one of the two most important bottlenecks we have to deal with (networking speed is the other).

Row Compression Effect:

No effect.

varchar(max)

Domain: ASCII characters, up to $2^{31} - 1$ characters (that is a maximum of 2 GB worth of text!).

Storage: There are a couple possibilities for storage based on the setting of the table option `'large value types out of row'`, which is set with the `sp_tableoption` system stored procedure:

- `OFF =`: The data for all the columns fits in a single row, and the data is stored in the row with the same storage costs for non-`max varchar` values. Once the data is too big to fit in a single row, data can be placed on more than one row. This is the default setting.
- `ON =`: You store `varchar(max)` values using 16-byte pointers to separate pages outside the table. Use this setting if the `varchar(max)` data will only seldom be used in queries.

Discussion:

The `varchar(max)` datatype is great replacement for the `text` datatype and all its quirks. You can deal with `varchar(max)` values using mostly the same functions and methods that you use with normal `varchar` values. There's a minor difference, though. As the size of your `varchar(max)` column grows toward the upper boundary, it's likely true that you aren't going to want to be sending the entire value back and forth over the network most of the time. I know that even on my 100 MB LAN, sending 2 GB is no instantaneous operation, for sure.

www.it-ebooks.info

There are a couple things to look at:

- The UPDATE statement has a .WRITE() clause to write chunks of data to the (max) datatypes. This is also true of varbinary(max).
- Unlike text and image values, (max) datatypes are accessible in AFTER triggers.

One word of warning for when your code mixes normal varchar and varchar(max) values in the same statement: normal varchar values do not automatically change datatype to a (max) type when the data being manipulated grows beyond 8,000 characters. For example, write a statement such as the following:

```
DECLARE @value varchar(max) = REPLICATE('X',8000) + REPLICATE('X',8000);
SELECT LEN(@value);
```

This returns the following result, which you would expect to be 16000, since you have two 8,000-character strings:

```
---------------
8000
```

The reason is that the type of the REPLICATE function is varchar, when replicating normal char values. Adding two varchar values together doesn't result in a varchar(max) value. However, most of the functions return varchar(max) values when working with varchar(max) values. For example:

```
DECLARE @value varchar(max) = REPLICATE(CAST('X' AS varchar(max)),16000);
SELECT LEN(@value);
```

This returns the following result:

```
----------------
16000
```

Row Compression Effect:

No effect.

text

Don't use the text datatype for any reason in new designs. It might not exist in the next version of SQL Server (though I have written that statement for several versions of this book). Replace with varchar(max) whenever you possibly can. See SQL Server Books Online for more information.

Unicode Character Strings: nchar, nvarchar, nvarchar(max), ntext

Domain: ASCII characters, up to $2^{15} - 1$ characters (2 GB of storage).

Storage: Same as other character datatypes, though every character takes 2 bytes rather than 1. (Note there is no support for any of the variable-length Unicode storage.)

www.it-ebooks.info

Discussion:

So far, the character datatypes we've been discussing have been for storing typical ASCII data. In SQL Server 7.0 (and NT 4.0), Microsoft implemented a new standard character format called Unicode. This specifies a 16-bit character format that can store characters beyond just the Latin character set. In ASCII—a 7-bit character system (with the 8 bits for Latin extensions)—you were limited to 256 distinct characters. This was fine for most English-speaking people but was insufficient for other languages. Asian languages have a character for each different syllable and are nonalphabetic; Middle Eastern languages use several different symbols for the same letter according to its position in the word. Unicode expanded the amount of characters and eliminated the need for code pages to allow for a vastly expanded character set (which allowed you to have multiple character sets in an 8-character encoding set in ASCII). SQL Server supports the Unicode Standard, version 3.2.

For these datatypes, you have the `nchar`, `nvarchar`, `nvarchar(max)`, and `ntext` datatypes. They are the same as the similarly named types (without the `n`) that we've already described, except for one thing: Unicode uses double the number of bytes to store the information, so it takes twice the space, thus cutting by half the number of characters that can be stored.

One quick tip: if you want to specify a Unicode value in a string, you append an `N` (must be a capital N, a lowercase will give you an error) to the front of the string, like so:

```
SELECT N'Unicode Value';
```

■ **Tip** You should migrate away from `ntext` as a datatype just as you should for the `text` datatype.

Row Compression Effect:

Just like their ASCII counterparts for fixed length types it will not store trailing blanks for the fixed-length types. As of SQL Server 2008R2, compression can compress Unicode values using what is known as the Standard Compression Scheme for Unicode (SCSU), which gives anywhere between 15 and 50 percent storage improvement depending on the character set. This is particularly interesting as a lot of third-party systems use Unicode storage "just in case," and it is becoming more and more the norm to use Unicode for pretty much everything in a system to allow for the future, even if you never make use of anything other than a standard ASCII character.

Binary Data

Binary data allows you to store a string of bytes. It's useful for storing just about anything, especially data from a client that might or might not fit into a character or numeric datatype. In SQL Server 2005, binary columns became even more useful, because you can use them when storing encrypted data. In Chapter 9, you'll learn about the encryption capabilities of SQL Server.

One of the restrictions of binary datatypes is that they don't support bitwise operators, which would allow you to do some powerful bitmask storage by being able to compare two binary columns to see not only whether they differ, but how they differ. The whole idea of the binary datatypes is that they store strings of bits. The bitwise operators can operate on integers, which are physically stored as bits. The reason for this inconsistency is fairly

www.it-ebooks.info

clear from the point of view of the internal query processor. The bitwise operations are operations that are handled in the processor, whereas the binary datatypes are SQL Server specific.

Binary literal values are specified as `0xB1B2B3 . . . BN`. `0x` tells you that it's a hexadecimal value. `B1` specifies the first single byte in hexadecimal.

binary[(length)]

Domain: Fixed-length binary data with a maximum length of 8,000 bytes.

Storage: Number of bytes the value is defined for. The default length is 1, if not specified (it is best practice to include a size).

Discussion:

The use of `binary` columns is fairly limited. You can use them to store any binary values that aren't dealt with by SQL Server. Data stored in `binary` is simply a string of bytes:

```
DECLARE @value binary(10) = CAST('helloworld' AS binary(10));
SELECT @value;
```

This returns the following result:

```
----------------------
0x68656C6C6F776F726C64
```

Now you can reverse the process:

```
SELECT CAST(0x68656C6C6F776F726C64 AS varchar(10));
```

This returns the following result:

```
----------
helloworld
```

Note that casting the value `HELLOWORLD` gives you a different value:

```
----------------------
0x48454C4C4F574F524C44
```

This fact that these two binary values are different, even for textual data that would be considered equivalent on a case-insensitive collation, has been one use for the `binary` datatype: case-sensitive searches. This is generally not the best way to do a case sensitive comparison, as it's far more efficient to use the `COLLATE` keyword and use a different collation if you want to do a case-insensitive comparison on string data.

www.it-ebooks.info

Row Compression Effect:

Trailing zeros are not stored but are returned when the values are used.

varbinary[(length)]

Domain: Variable-length binary data with a maximum length of 8,000 bytes.

Storage: Number of bytes the value is defined for, plus 2 bytes for variable-length overhead. The default length is 1, if not specified (it is a best practice to include a size).

Discussion:

The usage is the same as `binary`, except the number of bytes is variable.

Row Compression Effect:

No effect.

varbinary(max)

Domain: Binary data, up to $2^{31} - 1$ bytes (up to 2 GB for storage) when data is stored in SQL Server files, up to the max of the storage for data stored in the filestream. For more information and examples about the filestream, check Chapter 7.

Storage: There are a couple possibilities for storage based on whether the data is stored using the filestream setting, as well as the setting of the table option `'large value types out of row'`:

- `OFF =`: If the data for all the columns fits in a single row, the data is stored in the row with the same storage costs for non-`max varchar` values. Once the data is too big to fit in a single row, data can be placed on greater than one row.
- `ON =`: You store `varbinary(max)` values using 16-byte pointers to separate pages outside the table. Use this setting if the `varchar(max)` data will only seldom be used in queries.

Discussion:

The `varbinary(max)` datatype provides the same kinds of benefits for large binary values as the `varchar(max)` does for text. Pretty much you can deal with `varbinary(max)` values using the same functions and the same methods as you do with the normal `varbinary` values.

What's cool is that you can store text, JPEG and GIF images, and even Word documents and Excel spreadsheet data using the `varbinary(max)` type. On the other hand, it can be much slower and more programming work to use SQL Server as a storage mechanism for files, mostly because it's slow to retrieve really large values from the database as compared to from the file system. You can, however, use a filestream access to get the best of both possible worlds by using Win32 access to a file in a directory within the context of a transaction. This approach is described in greater detail in Chapter 7.

Row Compression Effect:

No effect.

www.it-ebooks.info

image

Just like the `text` datatype, the `image` datatype is being deprecated in this version of SQL Server. Don't use the `image` datatype in new designs if at all possible. It very well may not exist in the next version of SQL Server. Replace with `varbinary(max)` in any location you can. See SQL Server Books Online for more information or if you have existing image column data that you need to manipulate.

Other Datatypes

The following datatypes are somewhat less easy to categorize but are still commonly employed in OLTP systems:

- `bit`
- `rowversion` (`timestamp`)
- `uniqueidentifier`
- `cursor`
- `table`
- `sql_variant`

bit

Domain: `0`, `1`, or `NULL`.

Storage: A `bit` column requires 1 byte of storage per eight instances in a table. Hence, having eight `bit` columns will cause your table to be no larger than if your table had only a single `bit` column.

Discussion:

You use `bit` values as a kind of imitation Boolean value. A `bit` isn't a Boolean value, in that it has values `0` and `1`, not `True` and `False`. This is a minor distinction but one that needs to be made. You cannot execute code such as this:

```
IF (bitValue) DO SOMETHING;
```

A better term than a Boolean is a *flag*. A value of `1` means the flag has been set (such as a value that tells us that a customer does want e-mail promotions). Many programmers like to use character values `'yes'` or `'no'` for this, because this can be easier for viewing, but it can be harder to program with using built-in programming methods. In fact, the use of the `bit` datatype as a Boolean value has occurred primarily because many programming languages usually use `0` for `False` and nonzero for `True` (some use `1` or `-1` explicitly).

You can index a `bit` column, but usually it isn't of any value only to index it. Having only two distinct values in an index (technically three with `NULL`) makes for a poor index. (See Chapter 10 for more information about indexes. You may be able to use a filtered index to make some indexes on bit columns useful.) Clearly, a `bit` value most often should be indexed in conjunction with other columns.

Another limitation of the bit data type is that you can't do math operations or aggregates with bit columns. Math is somewhat expected, but there are certainly

www.it-ebooks.info

places where the MAX aggregate would be a very useful thing. You can cast the bit to a tinyint and use it in math/aggregates if you need to.

A relatively odd property of the bit datatype is that you can case the string values `'True'` and `'False'` to bit values 1 and 0 respectively. So the following will work:

```
SELECT CAST ('True' AS bit) AS True, CAST('False' AS bit) AS False;
```

Spelling counts (though not case) and other text values will.

Row Compression Effect:

Depends on the number of bits in the row. For a single bit value, a 4 bits will be needed because of the metadata overhead of compression.

■ **Tip** There's always a ton of discussion on the forums/newsgroups about using the `bit` datatype. It's often asked why we don't have a Boolean datatype. This is largely because of the idea that datatypes need to support `NULL` in RDBMSs, and a Boolean datatype would have to support `UNKNOWN` and `NULL`, resulting in four valued logic tables that are difficult to contemplate (without taking a long nap) and hard to deal with. So, we have what we have, and it works well enough.

---

rowversion (aka timestamp)

The `rowversion` datatype is a database-wide unique number. When you have a `rowversion` column in a table, the value of the `rowversion` column changes for each modification to each row in a 8-byte binary value. The value in the `rowversion` column is guaranteed to be unique across all tables in the datatype. It's also known as a `timestamp` value, but it doesn't have any time implications—it's merely a unique value to tell you that your row has changed.

■ **Tip** In the SQL standards, a `timestamp` datatype is equivalent to what you know as a `datetime` datatype. To avoid confusion, Microsoft has deprecated the name timestamp and now recommends that you use the name `rowversion` rather than `timestamp`, although you will notice that some of their examples and scripting tools will still reference the timestamp name.

The `rowversion` column of a table (you may have only one) is usually used as the data for an optimistic locking mechanism. The `rowversion` datatype is a mixed blessing. It's stored as an 8-byte `varbinary` value. Binary values aren't always easy to deal with, and their use depends on which mechanism you're using to access your data.

As an example of how the `rowversion` datatype works, consider the following batch:

```
SET NOCOUNT ON;

CREATE TABLE testRowversion
(
  value  varchar(20) NOT NULL,
  auto_rv  rowversion NOT NULL
);

INSERT INTO testRowversion (value) VALUES ('Insert');

SELECT value, auto_rv FROM testRowversion;
```


www.it-ebooks.info

```
UPDATE testRowversion
SET value = 'First Update';

SELECT value, auto_rv FROM testRowversion;

UPDATE testRowversion
SET value = 'Last Update';

SELECT value, auto_rv FROM testRowversion;
```

This batch returns the following results (your `auto_rv` column values may vary, but they should still be hexadecimal representations):

```
value                auto_rv
-------------        ------------------
Insert               0x00000000000007DA

value                auto_rv
--------             ------------------
First Update         0x00000000000007DB

value                auto_rv
-------------        ------------------
Last Update          0x00000000000007DC
```

You didn't touch the `auto_rv` column, and yet it incremented itself twice. However, you can't bank on the order of the `rowversion` values being sequential, because updates of other tables will change the value as well. All `rowversion` values in a database draw from the same pool of values. It's also in your best interest not to assume in your code that a `rowversion` number is an incrementing value. How `rowversion`s are implemented is a detail that will likely change in the future. If a better method of building database-wide unique values comes along that's even a hair faster, Microsoft will likely use it.

You can create variables of the `rowversion` type for holding `rowversion` values, and you can retrieve the last-used `rowversion` via the `@@dbts` configuration function. `Rowversion` columns are used in Chapter 11, where I demonstrate optimistic locking.

> *Row Compression Effect*:
>
> Uses an integer representation of the value, using 8 bytes. Then it can be compressed just like the `bigint` type.

Uniqueidentifier

Globally unique identifiers are fast becoming a mainstay of Microsoft computing. The name says it all—these identifiers are globally unique. According to the way that GUIDs are formed, there's a tremendously remote chance that there will ever be any duplication in their values as there are 2^{128} possible values. They're generated by a formula that includes the current date and time, a unique number from the CPU clock, and some other "magic numbers."

In your databases, these GUID values are stored in the `uniqueidentifier` type, which is implemented as a 16-byte binary value. An interesting use is to have a key value that's guaranteed to be unique across databases and servers. You can generate a GUID value in T-SQL using the `newid` function.

```
DECLARE @guidVar uniqueidentifier = NEWID();

SELECT  @guidVar AS guidVar;
```


www.it-ebooks.info

Returns (a similar value to) :

```
guidVar
------------------------------------
6C7119D5-D48F-475 C-8B60-50D0C41B6EBF
```

While GUIDs are stored as 16-byte binary values, they aren't exactly a straight binary value. You cannot put just any binary value into a `uniqueidentifier` column, because the value must meet the criteria for the generation of a GUID, which aren't exactly well documented. (For more information, a good resource is http://en.wikipedia.org/wiki/guid.)

If you need to create a `uniqueidentifier` column that's autogenerating, you can set a property in the `CREATE TABLE` statement (or `ALTER TABLE`, for that matter). It's the `ROWGUIDCOL` property, and it's used like so:

```
CREATE TABLE guidPrimaryKey
(
  guidPrimaryKeyId uniqueidentifier NOT NULL ROWGUIDCOL DEFAULT NEWID(),
  value varchar(10)
);
```

I've introduced a couple new things here: `rowguidcol` and default values. Suffice it to say that if you don't provide a value for a column in an insert operation, the default operation will provide it. In this case, you use the `NEWID()` function to get a new `uniqueidentifier`. Execute the following `INSERT` statement:

```
INSERT INTO guidPrimaryKey(value)
VALUES ('Test');
```

Then run the following command to view the data entered:

```
SELECT  *
FROM    guidPrimaryKey;
```

This returns the following result (though of course your key value will be different):

```
guidPrimaryKeyId                     value
------------------------------------ -----
490E8876-A695-4F5B-B53A-69109A28D493 Test
```

The `rowguidcol` property of a column built with the `uniqueidentifier` notifies the system that this is just like an identity column value for the table—a value with enforced uniqueness for a row in a table. Note that neither the identity nor the `rowguidcol` properties guarantee uniqueness. To provide such a guarantee, you have to implement your tables using `UNIQUE` constraints.

It would seem that the `uniqueidentifier` would be a better way of implementing primary keys, because when they're created, they're unique across all databases, servers, and platforms. However, there are two main reasons why you won't use `uniqueidentifier` columns to implement all your primary keys:

- *Storage requirements*: Because they're 16 bytes in size, they're considerably more bloated than a typical integer column.
- *Typeability*: Because there are 36 characters in the textual version of the GUID, it's hard to type the value of the GUID into a query, and it isn't easy to enter.

www.it-ebooks.info

If you're using the GUID values for the primary key of a table and you're clustering on this value, you can use another function to generate the values: `newSequentialId()`. You can use this function only in a default constraint. It's used to guarantee that the next GUID chosen will be greater than the previous value:

```
DROP TABLE guidPrimaryKey;
go
CREATE TABLE guidPrimaryKey
(
  guidPrimaryKeyId uniqueidentifier NOT NULL
          ROWGUIDCOL DEFAULT NEWSEQUENTIALID(),
  value varchar(10)
);
GO
INSERT INTO guidPrimaryKey(value)
VALUES  ('Test'),
        ('Test1'),
        ('Test2');
GO

SELECT  *
FROM    guidPrimaryKey;
```

This returns something like the following, with a notable progression to the values of the `guidPrimaryKeyId` column values:

```
guidPrimaryKeyId                        value
--------------------------------------  ------
18812704-49E3-E011-89D1-000 C29992276   Test
19812704-49E3-E011-89D1-000 C29992276   Test1
1A812704-49E3-E011-89D1-000 C29992276   Test2
```

You may notice that the increasing value appears to be in the letters to the far left. To the naked eye, it would appear that we could be pretty close to running out of values, since the progression of 18, 19, 1A is going to run out pretty quickly. The fact is, the values are not being sorted on the text representation of the GUID, but on the internal binary value.

Now, using a GUID for a primary key is just about as good as using an identity column for building a surrogate key, particularly one with a clustered index (they are still rather large at 16 bytes versus 4 for an `integer`, or even 8 for a `bigint`). That's because all new values will be added to the end of the index rather than randomly throughout the index. (Chapter 10 covers indexes, but be cognizant that a random value distributed throughout your rows can cause fragmentation unless you provide a fill factor that allows for adding rows to pages.) Values in the `uniqueidentifier` type will still be four times as large as an `integer` column, hence requiring four times the storage space. This makes using a `uniqueidentifier` a less than favorable index candidate from the database storage layer's perspective. However, the fact that it can be generated by any client and be guaranteed unique is a major plus, rather than requiring you to generate them in a single threaded manner to ensure uniqueness.

Row Compression Effect:

No effect.

www.it-ebooks.info

cursor

A cursor is a mechanism that allows row-wise operations instead of using the normal set-wise way. You use the `cursor` datatype to hold a reference to a SQL Server T-SQL cursor. You may not use a `cursor` datatype as a column in a table. Its only use is in T-SQL code to hold a reference to a cursor, which can be passed as a parameter to a stored procedure.

Row Compression Effect:

No effect.

table

The `table` type is kind of two different things now in 2008. First you have the `table` type that is essentially a temporary table that you can declare like a variable at runtime, and you can define its characteristics. Second (and new to 2008), you have table types that are defined and stored for later use, for example, as table-valued parameters. I have broken these two different types of uses down into two sections. Neither usage is affected by row compression.

Table Variables

The `table` variable has a few things in common with the `cursor` datatype, but instead of a cursor, it holds a reference to a result set. The name of the datatype is a pretty bad choice, because it will make functional programmers think that they can store a pointer to a table. It's actually used to store a result set as a temporary table. In fact, the table is exactly like a temporary table in implementation. However, you don't get any kind of statistics on the table, nor are you able to index the `table` datatype, other than to apply `PRIMARY KEY` and `UNIQUE` constraints in the table declaration. You can also have `CHECK` and `DEFAULT` constraints.

Unlike local temporary tables (those declared with # preceding the name), `table` datatype variables won't cause recompiles in stored procedures that use them, because they don't have any statistics to change the plan anyway. Use them only for modestly small sets of data (hundreds of rows, not thousands, generally), such as when all the data in the table can fit on a single data page.

The following is an example of the syntax needed to employ the `table` variable type:

```
DECLARE @tableVar TABLE
(
  id int IDENTITY PRIMARY KEY,
  value varchar(100)
);
INSERT INTO @tableVar (value)
VALUES ('This is a cool test');

SELECT id, value
FROM @tableVar;
```

This returns the following result:

```
id          value
---         -------------------
1           This is a cool test
```


www.it-ebooks.info

As with the cursor datatype, you may not use the table datatype as a column in a table, and it can be used only in T-SQL code to hold a set of data. One of the primary purposes for the table datatype is for returning a table from a user-defined function, as in the following example:

```
CREATE FUNCTION table$testFunction
(
  @returnValue varchar(100)
)
RETURNS @tableVar table
(
   value varchar(100)
)
AS
BEGIN
  INSERT INTO @tableVar (value)
  VALUES (@returnValue);
  RETURN;
END;
```

Once created, you can use the table datatype returned by the function using typical SELECT syntax:

```
SELECT *
FROM dbo.table$testFunction('testValue');
```

This returns the following result:

```
value
---------
testValue
```

One interesting thing about the table datatype is that it isn't subject to transactions. For example:

```
DECLARE @tableVar TABLE
(
  id int IDENTITY,
  value varchar(100)
);
BEGIN TRANSACTION;

INSERT INTO @tableVar (value)
VALUES ('This will still be there');

ROLLBACK TRANSACTION;

SELECT id, value
FROM @tableVar;
```

This returns the following result:

```
id          value
---         ----------------------
1           This will still be there
```


www.it-ebooks.info

For this reason, these tables are useful for logging errors, because the data is still available after the ROLLBACK TRANSACTION.

Table Valued Parameters

One of the oft-requested features for SQL Server was the ability to pass in a table of values to a stored procedure. Using the table type, you can now do this, but not in as free a manner as you probably would have initially hoped. Instead of being able to define your table on the fly, you are required to use a type that you predefine.

The table type you will define is the same as the datatype alias we discussed in Chapter 5, except you specify an entire table, with all of the same things that a table variable can have, including PRIMARY KEY, UNIQUE, CHECK, and DEFAULT constraints.

An example that I imagine will be very commonly imitated is the generic table type with a list of integer values to pass as a parameter or to use in a query instead of an IN clause:

```
CREATE TYPE GenericIdList AS TABLE
(
  Id Int Primary Key
);
```

You declare the table variable just like any other and then load and use the variable with data just like any other local variable table:

```
DECLARE @ProductIdList GenericIdList;

INSERT  INTO @productIDList
VALUES  (1),(2),(3),(4);

SELECT  ProductID, Name, ProductNumber
FROM    AdventureWorks2012.Production.Product
         JOIN @productIDList AS list
          ON Product.ProductID = List.Id;
```

This returns the following:

```
ProductID         Name                     ProductNumber
---------         ---------------------    -------------
1                 Adjustable Race          AR-5381
2                 Bearing Ball             BA-8327
3                 BB Ball Bearing          BE-2349
4                 Headset Ball Bearings    BE-2908
```

Of course, you can then use the type in your stored procedure creation statements as well:

```
CREATE PROCEDURE product$list

(
  @productIdList GenericIdList READONLY
)
AS
SELECT  ProductID, Name, ProductNumber
FROM    AdventureWorks2008R2.Production.Product
          JOIN @productIDList AS List
            ON Product.ProductID = List.Id;
```


www.it-ebooks.info

Unfortunately, you cannot pass a set of row constructors to the stored procedure; instead, you will need to declare and load a table variable to use this construct from T-SQL.

```
DECLARE @ProductIdList GenericIdList;

INSERT  INTO @productIDList
VALUES  (1),(2),(3),(4);

EXEC    product$list @ProductIdList;
```

What makes this really nice is that in ADO.NET, you can declare a `DataTable` object and pass it to the procedure as a parameter, just like any other value now. This will make the ability to insert multiple items at a time or `SELECT` multiple rows far easier than ever before. In the past, we used a kludgy, comma-delimited list or XML to do this, and it worked, but not in a natural manner we are accustomed to, and it was generally slow. This method will now work in a natural manner, allowing us to finally support multiple operations in a single transaction from an easy-to-build ADO.NET construct.

sql_variant

The catchall datatype, the `sql_variant` type, allows you to store a value of almost any datatype that I've discussed. This ability allows you to create a column or variable where you don't know ahead of time exactly what kind of data will be stored. The `sql_variant` datatype allows you to store values of various SQL Server–supported datatypes, except for `varchar(max)`, `varbinary(max)`, `xml`, `text`, `ntext`, `rowversion/timestamp`, and `sql_variant`.

■ **Note** Although the `rowversion` datatype cannot be stored directly in a `sql_variant`, a `rowversion` value can be stored in a `binary(8)` variable, which can in turn be stored in a `sql_variant` variable. Also, it might seem strange that you can't store a variant in a variant, but this is just saying that the `sql_variant` datatype doesn't exist as such—SQL Server chooses the best type of storage in which to store the value you give to it.

Generally, `sql_variant` is a datatype to steer clear of unless you really cannot know the datatype of a given value until the user enters the value. I used the `sql_variant` in Chapter 8 when I implemented the user-specified data storage using the entity-attribute-value solution. This allowed the user to enter any type of data and then have the system store the data in the most appropriate method.

The `sql_variant` type has some obvious value, and I used it earlier in Chapter 8 when building an entity-attribute-value solution for an open schema solution. By not needing to know the type at design time, you can allow the user to insert any type of data that they might want.

However, the positives lead directly to the negatives to the `sql_variant` type. Although simple storage and viewing of the data isn't too hard, it isn't easy to manipulate data once it has been stored in a `sql_variant` column. I'll leave it to you to read the information fully in the parts of SQL Server Books Online that deal with `variant` data, but some issues to consider are as follows:

- *Difficulties assigning data from a* `sql_variant` *column to a stronger typed datatype*: You have to be careful, because the rules for casting a variable from one datatype to another are difficult and might cause errors if the data can't be cast. For example, you can't cast the `varchar(10)` value `'Not a Date'` to a `datetime` datatype. Such problems become an issue when you start to retrieve the `variant` data out of the `sql_variant` datatype and try to manipulate it.
- `NULL` `sql_variant` *values are considered to have no datatype*: Hence, you'll have to deal with `sql_variant` `NULL`s differently from nulls in other datatypes.

www.it-ebooks.info

- *Comparisons of* `variant`*s to other datatypes could cause difficult-to-catch programmatic errors, because of the* `sql_variant` *value instance's datatype*: Usually, the compiler will know whether you try to run a statement that compares two incompatible datatypes, such as `@intVar = @varcharVar`. However, if the two variables in question were defined as `sql_variant`s and the datatypes don't match, then the values won't compare because of the datatype incompatibilities.

When working with `sql_variant` variables or columns, you can use the `SQL_VARIANT_PROPERTY` function to discover the datatype of a given `sql_variant` value. For example:

```
DECLARE @varcharVariant sql_variant = '1234567890';

SELECT  @varcharVariant AS varcharVariant,
        SQL_VARIANT_PROPERTY(@varcharVariant,'BaseType') as baseType,
        SQL_VARIANT_PROPERTY(@varcharVariant,'MaxLength') as maxLength,
        SQL_VARIANT_PROPERTY(@varcharVariant,'Collation') as collation;
```

The preceding statement returns the following result:

```
varcharVariant      baseType        maxLength     collation
--------------      --------        ---------     ----------------------------
1234567890          varchar         10            SQL_Latin1_General_CP1_CI_AS
```

For numeric data, you can also find the precision and scale:

```
DECLARE @numericVariant sql_variant = 123456.789;

SELECT @numericVariant AS numericVariant,
  SQL_VARIANT_PROPERTY(@numericVariant,'BaseType') as baseType,
  SQL_VARIANT_PROPERTY(@numericVariant,'Precision') as precision,
  SQL_VARIANT_PROPERTY(@numericVariant,'Scale') as scale;
```

This returns the following result:

```
numericVariant        baseType        precision        scale
--------------        --------        ---------        -----
123456.789            numeric         9                3
```

Not Simply Scalar Datatypes

This section will deal with the class of datatypes that have been implemented by Microsoft that aren't really scalar values. Another common term for these datatypes that have cropped up around the Internet is *beyond relational*, but to many people this is a confusing term. In one way of thinking, these are perfectly scalar types, but in yet another they really aren't.

The non-scalar types include the following:

- `hierarchyId`: Used to help build and manage a tree structure. It is very close to being a scalar type with several methods that can be applied to traverse and work with a hierarchy.

www.it-ebooks.info

- *Spatial types*: `geometry` for dealing with planar/Euclidean (flat-Earth) data; `geography` for ellipsoidal (round-Earth) data, such as GPS longitude and latitude data. The spatial types technically hold arrays of values that represent sets on their own (and as you will see, you can join two shapes to see whether they overlap).
- `XML`: Used to store and manipulate XML values. A single XML column can more or less implement a database almost on its own.

Each of these types has some value to someone and fills a void that cannot be straightforwardly represented with the relational model, at least not as easily. I am a prude in many ways when it comes to normalization, but not every situation calls for strict adherence to the first normal form. What is, however, important is to know what you are doing and how you are violating the normal forms when you do and when it is appropriate.

www.it-ebooks.info

APPENDIX B

DML Trigger Basics and Templates

In this appendix, I briefly introduce triggers and how they are written and present templates that you can use as the basis for creating DML triggers for all sorts of uses. There are two different types of DML triggers that are coded very similarly, but serve different purposes:

- The `AFTER` trigger is made to be called after an `INSERT`, `UPDATE`, or a `DELETE`, and the constraints it references, is called. It is used to apply complex checks and to add side effects to a DML operation as may be needed. `AFTER` triggers are usually used for handling rules that won't fit into the mold of a constraint, for example, rules that require data to be stored, such as a logging mechanism. You may have a virtually unlimited number of `AFTER` triggers that fire on `INSERT`, `UPDATE`, and `DELETE`, or any combination of them.
- The `INSTEAD OF` type of trigger is called before a DML operation, even before constraints. The actual operation on the table is not done, and if you want the data to go in the table, you have to re-perform the operation. In this way, you can do whatever you want with the data, either doing exactly what was requested by the user or doing something completely different (you can even just ignore the operation altogether). You can have a maximum of one `INSTEAD OF INSERT`, `UPDATE`, and `DELETE` trigger of each type per table. It is allowed, but not a generally good idea, to combine all three into one and have a single trigger that fires for all three operations. Instead of triggers can also be used on views to make any view updatable.

Coding DML triggers is very much like coding a stored procedure, with some important differences. Instead of having data passed in to the stored procedure, you use two special in-memory tables that are instantiated for the life of the operation and are scoped specifically to code executing directly in the trigger. These tables are named `inserted` and `deleted`. The `inserted` table contains new or updated rows for an `INSERT` or `UPDATE` operation, and the `deleted` table contains the row values from the affected rows as they were before the DML operation that have been deleted for a `DELETE` statement execution or that have been modified by an `UPDATE` statement. It is important to note that these tables will have multiple rows in them if you modify more than one row in your DML statement. Also, since they are scoped to the executing trigger, if you call a stored procedure, the tables will not be accessible, and if your trigger causes another trigger to fire, the contents of the tables will be for the currently executing trigger.

This appendix introduces a set of templates that are used in several chapters of this book, as well as providing fairly exhaustive examples of using the template. (In Chapter 7, I present the more "realistic" uses of triggers, while here in this appendix I am trying to show some of the more esoteric ways triggers can be used.)

www.it-ebooks.info

While triggers are similar to procedures, there are some fairly major differences that are important. In the following bulleted list are overviews of some of the concepts you need to be cognizant of when writing triggers, including some settings that can be very useful:

- *Multi-row considerations*: Triggers fire once for a DML operation, regardless of how many rows are affected. Hence, statements within triggers have to be coded with multiple rows in mind. This can be confusing, because unlike what seems to be natural, trigger code for typically needs to look for rows that don't meet your criteria, instead of those that do. Unless you want to force users into entering one row at a time, you have to code your triggers in a way that recognizes that more than one row in the table might be being modified.
- *Performance*: When a reasonable number of rows are dealt with in a trigger (certainly any amount that are part of typical OLTP operations), they are usually quite fast, but as the number of modified rows increases into the thousands, triggers can become tremendous performance drains. This is largely because the `inserted` and `deleted` tables aren't "real" tables so they don't have indexes that the optimizer can use to optimize for. Hence, the plans chosen for the queries can be fairly optimistic about the number of rows in `inserted` and `deleted` tables. As said, because OLTP systems usually deal with small numbers of rows at a time, there's rarely a major performance hit because of using triggers, but it is something you have to be cognizant of anytime you create a new trigger and particularly when you have to do a large data load/purge.
- *Determining modified columns*: For performance reasons, you may not want to validate data that's in a column that isn't affected by a DML statement. You can tell which columns were part of the `INSERT` or `UPDATE` statement by using the `UPDATE(<columnName>)` function to check the column to see whether it was involved in the DML operation. Note that this does not indicate that a value has *changed*, just that the column was referenced. For example, given the simple statement `UPDATE tableName SET column1 = column1`, the values would not change, but `UPDATE(column1)` would return true. (There is also another method using the function `COLUMNS_UPDATED(columnBitmask)` to check the columns by their position in the table. `1 + 2 + 4 = 7` would mean the first three columns were updated, but it's generally a bad practice to address columns in a table positionally, for future maintenance purposes.)
- *Having multiple* `AFTER` *triggers for the same action*: It's possible to have many different triggers on a table, which gives you the ability to add triggers to third-party systems without touching triggers that the third-party created. However, often the order of triggers can be important, especially when you have to deal with validating data that another trigger might modify. You do get some minor control over the order in which triggers fire. Using the `sp_settriggerorder` system stored procedure you can choose the first and the last trigger to fire. Usually this is all you need, because there are places where you want to set the first trigger (often the third-party trigger) and the last trigger (such as a trigger to implement an audit trail, as we do in a later section).
- *Nesting triggers*: Take care when building `AFTER` triggers that modify data (the same table or other tables) because these updates could in turn cause other triggers to fire. `INSTEAD OF` triggers always cause other triggers to fire. I am not going to go into deep detail with the concerns with nesting triggers, but it is important to know that unlike any DML statement, DML in triggers may or may not cause additional triggers to fire based on the following settings. Make sure that you test your triggers and settings to ensure that what you expect to occur does occur. There are two important settings to be concerned with:

www.it-ebooks.info

    - *Server option*—`sp_serveroption`—*nested triggers*: When this setting is set to `1`, it indicates that if you modify a different table, that table's trigger will be fired. This setting is usually set to `1`, because it allows for data validations to occur in the other tables without coding every business rule again.
    - *Database option*—`ALTER DATABASE-RECURSIVE_TRIGGERS`: When set to `ON`, when an `AFTER` trigger modifies the data in the same table, the triggers for that table execute again. This setting is usually set to `OFF`. Because it's common practice to modify the same table in the trigger, it's assumed that any modifications done in a trigger will meet all business rules for the same table.
- *Server option*—`sp_serveroption`—*disallow results from triggers*: It is a very bad practice to return results from a trigger during production code. However, when debugging a trigger it can be a very useful practice. Turn this setting on will ensure that any trigger that tries to return data to the client will get the following error message: `Msg 524, Level 16, State 1, Procedure test$InsertTrigger, A trigger returned a resultset and the server option 'disallow results from triggers' is true.`

Because multi-row operations are the most frequently messed up aspect of trigger writing, it's worth discussing this aspect in more detail. If you insert a thousand rows, the `inserted` table will have a thousand rows. The `deleted` table will remain empty on an insert. When you delete rows, the `deleted` table is filled, and the `inserted` table remains empty. For an `UPDATE`, both tables are filled with the rows in the updated table that had been modified as they appeared before and after the update.

Because of this, writing validations must take this into consideration. For example, the following all-too-typical approach wouldn't be a good idea:

```
SELECT @column1 = column1 FROM inserted;
IF @column1 < 0
  BEGIN
    --handle the error
```

This is wrong because only a single row would be checked—in this case, the last row that the `SELECT` statement came to. (There's no order, but `@column1` would be set to every value in the `inserted` table and would end up with the last value it came upon.) Instead, the proper way to code this (assuming column1 does not allow nulls) would be as follows:

```
If EXISTS (SELECT *
           FROM  inserted
           WHERE  column1 < 0)
   BEGIN
       --handle the error
```

This works because each row in the `inserted` table is checked against the criteria. If any rows do match the criteria, the `EXISTS` Boolean expression returns `True`, and the error block is started.

You'll see this more in the example triggers. However, you need to make a conscious effort as you start to code triggers to consider what the effect of modifying more than one row would be on your code, because you certainly don't want to miss an invalid value because of coding like the first wrong example.

If you need a full reference on the many details of triggers, refer to SQL Server Books Online. In the following section, we'll look at the different types of triggers, the basics of coding them, and how to use them to handle the common tasks for which we use triggers. Luckily, for the most part triggers are straightforward, and the basic settings will work just fine.

www.it-ebooks.info

■ **Note** I don't think I could stress nearly enough about the need to understand multi-row operations in triggers. Almost every time a question is raised on the forums about triggers, the code that gets posted contains code that will handle only one row.

Triggers often get a bad name because they can be pretty quirky, especially due to the fact that they can kill performance when you are dealing with large updates. For example, if you have a trigger on a table and try to update a million rows, you are likely to have issues. However, for most OLTP operations in a relational database, operations shouldn't be touching more than a handful of rows at a time. Trigger usage does need careful consideration, but where they are needed, they are terribly useful. My recommendation is to use triggers when you need to do the following:

- Perform cross-database referential integrity
- Check inter-row rules, where just looking at the current row isn't enough for the constraints
- Check inter-table constraints, when rules require access to data in a different table
- Introduce desired side effects to your data-modification queries, such as maintaining required denormalizations
- Guaranteeing that no insert, update, or delete operations can be executed on a table, even if the user does have rights to perform the operation

Some of these operations could also be done in an application layer, but for the most part, these operations are far easier and safer, particularly for data integrity, when done automatically using triggers. When it comes to data protection, the primary advantage that triggers have over constraints is the ability to access other tables seamlessly and to operate on multiple rows at once. In a trigger, you can run almost every T-SQL command, except for the following:

ALTER DATABASE	RECONFIGURE
CREATE DATABASE	RESTORE DATABASE
DROP DATABASE	LOAD LOG
RESTORE LOG	LOAD DATABASE

Also, you cannot use the following commands on the table that the trigger protects:

CREATE INDEX	ALTER PARTITION FUNCTION
ALTER INDEX	DROP TABLE
DROP INDEX	ALTER TABLE
DBCC REINDEX	

It wouldn't be a very good design to change the schema of *any* table or do any of the things in this list in a trigger anyhow, much less the one that the trigger is built on, so these aren't overly restrictive requirements at all.

When using triggers, it is important to keep them as lean as possible. Avoid using cursors, calling stored procedures, or doing any sort of looping operation, and instead get the job done fast. If you need to do some extra processing, such as sending an e-mail for every row affected by the trigger, create a table that can be used as a queue for another process to work on. When your code is executing in a trigger, you can be holding locks,

www.it-ebooks.info

unnecessarily forcing other users to wait, and you cannot be completely certain that the rows that were modified to fire the trigger will actually be committed, and if it gets rolled back, you probably won't want something like an e-mail sent. If you send the e-mail directly via the trigger, it may have already sent the e-mail by the time the transaction is rolled back.

■ **Note** In SQL Server 2005 and later, the SQL-based mail object was changed to use Service Broker to implement mail, so if you roll back the transaction, it will roll back the mail command, but you would still have to use a cursor to call the send mail procedure rather than letting that occur asynchronously.

Error handling for triggers will use the simple error logging procedure that is implemented in Chapter 12 as part of the "Database Template Objects" section. It will use the following code (it is repeated in the downloads for this appendix):

```
--[Error logging section]
DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
        @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
        @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE()
EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;
```

It captures the error values from the `ERROR_` system functions values that are populated after an error occurs, and then calls the procedure to write the data to a table named `Utility.ErrorLog`. This can be commented out if you don't car to log errors, and it can also be used in any of your code to capture errors.

AFTER Triggers

`AFTER` triggers fire after all the constraints pass all constraint requirements. For instance, it wouldn't be useful to insert rows in a child table, causing its entire trigger/constraint chain to fire, when the far cheaper operation of checking a foreign key reference or a check constraint might fail the operation. Equally, you wouldn't want to check the status of all the rows in your table until you've completed all your changes to them; the same could be said for cascading delete operations.

All triggers I write in this book will use a common template that sets up the code that is used over and over again (and is pretty tedious to set up the first time anyhow). The template that I use follows. I have done my best to comment the code here and in all uses, so it will be easy for you to reuse. I personally use this for all triggers I create as well.

```
CREATE TRIGGER <schema>.<tablename>$<actions>[<purpose>]Trigger
ON <schema>.<tablename>
AFTER <comma delimited actions> AS
BEGIN
  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  DECLARE @msg varchar(2000),  --used to hold the error message
```


www.it-ebooks.info

```
    --use inserted for insert or update trigger, deleted for update or delete trigger
    --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
    --that is equal to number of merged rows, not rows being checked in trigger
        @rowsAffected int = (SELECT COUNT(*) FROM inserted);
    --      @rowsAffected int = (SELECT COUNT(*) FROM deleted);

    --no need to continue on if no rows affected
    IF @rowsAffected = 0 RETURN;
    BEGIN TRY
       --[validation section]
       --[modification section]
    END TRY
    BEGIN CATCH
     IF @@trancount > 0
       ROLLBACK TRANSACTION;

       --[Error logging section]
     DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
               @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
               @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE();
     EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;

     THROW; --will halt the batch or be caught by the caller's catch block

 END CATCH
END
```

■ **Tip** The `AFTER` keyword was introduced in the 2000 version of SQL Server when `INSTEAD OF` triggers were introduced. Prior to this, the keyword was `FOR`, since the trigger was for certain actions. Both are still quite common, but it is best to use `AFTER` in all new code.

I generally write triggers so that when the first error occurs, an error will be raised, and I roll back the transaction to halt any further commands. In the trigger template, there are three areas where code is added, denoted by comments headers:

- `--[validation section]`: In this section you will add validation logic that will be executed after the DML has been performed on the table. This would be used instead of a constraint when you need to code some complex validation that doesn't fit the mold of a constraint well.
- `--[modification section]`: Used for DML statements to modify the contents of tables or to do some other operation that has side effects of some sort. In this section, you might modify the same table as the triggering table or any other table.
- `--[Error logging section]`: This is the part of the trigger where you log any errors, either by inserting them into a table, or to the error log using `xp_logevent`, In the trigger templates, I use the `Utility.ErrorLog$Insert` procedure we created in Chapter 12 that writes to the `Utility.ErrorLog` table.

The form I use for almost every `[validation section]` I build is similar to the following code. I typically try to code different messages for the case when one row was affected by the DML operation by checking

www.it-ebooks.info

the @rowsAffected variable that is set earlier in the trigger by checking the number of rows in the inserted or deleted tables. This allows for better error messages (like to include the invalid value) for more typical singleton case, and a more generic explanation when many rows were changed. (If you commonly get errors in multi-row operations, you can enhance the multi-row error message to use an aggregate to return a single invalid value, but this can be costly for large updates.

```
IF EXISTS (<Boolean condition, commonly using inserted and/or deleted tables>)
   BEGIN
      IF @rowsAffected = 1 --custom error message for single row
            SELECT @msg = CONCAT('<reason>', inserted.value)
            FROM  inserted; --and/or deleted, depending on action
      ELSE
         SELECT @msg = '<more generic reason>';

      --in the TRY . . . CATCH block, this will redirect to the CATCH
      THROW 50000, @msg, 16;
   END;
```

The [modification section] section in SQL Server 2005 and later can simply be set up as simple INSERT, UPDATE, or DELETE statements thanks to the TRY-CATCH construct. Any errors raised because of the DML, such as from a constraint or another trigger, will be caught and sent to a CATCH block. In the CATCH block, I use a procedure called utility.ErrorLog$insert to log the error for later debugging, which was mentioned earlier.

It is necessary here to provide further explanation about how I capture the number of rows affected. In earlier editions of the book, I simply captured the output of @@rowcount. Until SQL Server 2005, this was sufficient. In 2008, the MERGE statement changed that (and in the last edition of the book I got it wrong) because every action clause in the MERGE statement calls the trigger, and the @@rowcount value is the one from the MERGE call, not the individual clauses. So instead of @@rowcount, I use the following (comments from trigger removed for clarity):

```
DECLARE @msg varchar(2000),  --used to hold the error message
        @rowsAffected int = (SELECT COUNT(*) FROM inserted)
        --@rowsAffected int = (SELECT COUNT(*) FROM deleted)
        ;
```

For the insert and update triggers, I count rows in inserted, and for the delete operation, it will use the deleted virtual table. As an example of the issue, I take the following simple table (In the downloaded code, I put this code in a new database called TriggerDemo:

```
CREATE TABLE test
(
   testId int
);
```

And I add the following simple trigger that covers all actions. All the trigger does is to output a row that tells us the value of @@rowcount global variable, the number of rows in each of the virtual table, and then a calculated value that tells you if the operation was an insert, update, or a delete.

```
CREATE TRIGGER test$InsertUpdateDeleteTrigger
ON test
AFTER INSERT, UPDATE, DELETE AS
BEGIN
         DECLARE @rowcount int = @@rowcount,  --stores the number of rows affected
                 @rowcountInserted int = (SELECT COUNT(*) FROM inserted),
                 @rowcountDeleted int = (SELECT COUNT(*) FROM deleted);
```


www.it-ebooks.info

```
        SELECT  @rowcount AS [@@rowcount],
                @rowcountInserted AS [@rowcountInserted],
                @rowcountDeleted AS [@rowcountDeleted],
                       CASE WHEN @rowcountInserted = 0 THEN 'DELETE'
                          WHEN @rowcountDeleted = 0 THEN 'INSERT'
                                ELSE 'UPDATE' END AS Operation;
END;
```

Remember, if you have set the disallow results from triggers; you will need to turn that off for this to work. Be sure and turn it back on after our demonstrations, as while this configuration value is an excellent way to allow yourself to use results for testing, in practice you do not want to have results returned from your triggers.

```
EXEC sp_configure 'show advanced options',1;
RECONFIGURE;
GO
EXEC sp_configure 'disallow results from triggers',0;
RECONFIGURE;
```

So now, let's add two rows to the table:

```
INSERT INTO test
VALUES (1),
       (2);
```

From the trigger, you will see the following output:

```
@@rowcount      @rowcountInserted       @rowcountDeleted        Operation
----------      -----------------       ----------------        ---------
2               2                       0                       INSERT
```

As expected, `@@rowcount` was 2, and the count of rows from the inserted table is also 2. Now, execute this simple MERGE that uses a CTE that will give us three operations, a delete for `testId` = 1, an update for `testId` = 3, and an insert for `testId` = 3.

```
WITH  testMerge AS (SELECT *
                    FROM  (VALUES(2),(3)) AS testMerge (testId))
MERGE test
USING (SELECT testId FROM testMerge) AS source (testId)
       ON (test.testId = source.testId)
WHEN MATCHED THEN
        UPDATE SET testId = source.testId
WHEN NOT MATCHED THEN
        INSERT (testId) VALUES (source.testId)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

Now you will see the following output, which appears as if 9 rows have been modified, even though what actually happened was that 1 row was inserted, 1 rows was updated, and 1 more row was deleted:

```
@@rowcount      @rowcountInserted       @rowcountDeleted       Operation
----------      -----------------       ----------------       ---------
3               1                       0                      INSERT
```


www.it-ebooks.info

```
@@rowcount      @rowcountInserted       @rowcountDeleted       Operation
----------      -----------------       ----------------       ---------
3               1                       1                      UPDATE

@@rowcount      @rowcountInserted       @rowcountDeleted       Operation
----------      -----------------       ----------------       ---------
3               0                       1                      DELETE
```

In my normal triggers, I generally do not mix any DML operations (so I have an insert trigger, and update trigger, and a separate delete trigger), and if I used the `@rowcount` value for anything (such as we usually quit the trigger if 0 rows were affected), we would have treated each trigger as having multiple rows. In either case, in this trigger, I had three calls with a single row modification, but `@@rowcount` indicated three rows.

In Chapter 7, `AFTER` triggers will be used to implement triggers to solve realistic the following sort of problems:

- *Range checks on multiple rows*: Make sure that a summation of values of column, usually over some grouping, is within some specific range of values.
- *Maintaining summary values (only as necessary)*: Basically updating one value whenever one or more values change in a different table.
- *Cascading inserts*: After a row is inserted into a table, one or more other new rows are automatically inserted into other tables. This is frequently done when you need to initialize a row in another table, quite often a status of some sort.
- *Child-to-parent cascades*: Performing cascading operations that cannot be done using a typical foreign key constraint.
- *Maintaining an audit trail*: Logging changes made to a table in the background.
- *Relationships that span databases and servers*: Basic referential integrity only works within the confines of a database.

In this appendix, I will present an extended example that demonstrates some of the power and uses of triggers. I create a trigger than makes sure that data, grouped on a given value always stays > 0. The triggers in this example are some of the most complex triggers I have written (since the advent of constraints, at least), and I did this to make the trigger a bit complex to show some of the power of triggers that are seldom needed (but certainly interesting when needed).

I take the data from a table called `Example.AfterTriggerExample`, which has a simple integer key; a column called `GroupingValue`, which serves as a kind of Account to group on; and `Example.AfterTriggerExampleGroupingBalance`, which holds the running balance.

```
CREATE SCHEMA Example;
GO
--this is the "transaction" table
CREATE TABLE Example.AfterTriggerExample
(

    AfterTriggerExampleId int CONSTRAINT PKAfterTriggerExample PRIMARY KEY,
    GroupingValue     varchar(10) NOT NULL,
    Value             int NOT NULL

);
GO
```


www.it-ebooks.info

```
--this is the table that holds the summary data
CREATE TABLE Example.AfterTriggerExampleGroupBalance
(

    GroupingValue varchar(10) NOT NULL
         CONSTRAINT PKAfterTriggerExampleGroupBalance PRIMARY KEY,
    Balance    int NOT NULL
);
```

Then I create the following insert trigger. The code is commented as to what is occurring in there. There are two major sections, one to validate data, which does a summation on the items in the table and makes sure the sums are greater than 0, and another to write the denormalization/summary data. I could have implemented the non-negative requirement by putting a constraint on the `Example.AfterTriggerExampleGroupBalance` to require a `Balance >= 0`, but I am trying to show triggers with a validation and a cascading/modification action.

```
CREATE TRIGGER Example.AfterTriggerExample$InsertTrigger
ON Example.AfterTriggerExample
AFTER INSERT AS
BEGIN

  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  DECLARE @msg varchar(2000),  --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
      @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  --      @rowsAffected int = (SELECT COUNT(*) FROM deleted);

  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  BEGIN TRY
        --[validation section]
        --Use a WHERE EXISTS to inserted to make sure not to duplicate rows in the set
        --if > 1 row is modified for the same grouping value
        IF EXIStS (SELECT AfterTriggerExample.GroupingValue
             FROM  Example.AfterTriggerExample
             WHERE EXISTS (SELECT *
             FROM Inserted
             WHERE AfterTriggerExample.GroupingValue =
                           Inserted.Groupingvalue)
                  GROUP BY AfterTriggerExample.GroupingValue
                  HAVING SUM(Value) < 0)
       BEGIN
          IF @rowsAffected = 1
             SELECT @msg = CONCAT('Grouping Value "', GroupingValue,
                  '" balance value after operation must be greater than 0')
             FROM  inserted;
          ELSE
             SELECT @msg = CONCAT('The total for the grouping value must ',
                         'be greater than 0');
          THROW 50000, @msg, 16;
       END;
```


www.it-ebooks.info

```
    --[modification section]
    --get the balance for any Grouping Values used in the DML statement
    WITH GroupBalance AS
    (SELECT AfterTriggerExample.GroupingValue, SUM(Value) AS NewBalance
     FROM  Example.AfterTriggerExample
     WHERE EXISTS (SELECT *
                   FROM Inserted
                   WHERE AfterTriggerExample.GroupingValue = Inserted.Groupingvalue)
     GROUP BY AfterTriggerExample.GroupingValue )

    --use merge because there may not be an existing balance row for the grouping value
    MERGE Example.AfterTriggerExampleGroupBalance
    USING (SELECT GroupingValue, NewBalance FROM GroupBalance)
                         AS source (GroupingValue, NewBalance)
    ON  (AfterTriggerExampleGroupBalance.GroupingValue = source.GroupingValue)
    WHEN MATCHED THEN --a grouping value already existed
        UPDATE SET Balance = source.NewBalance
    WHEN NOT MATCHED THEN --this is a new grouping value
        INSERT (GroupingValue, Balance)
        VALUES (Source.GroupingValue, Source.NewBalance);
  END TRY
  BEGIN CATCH
   IF @@trancount > 0
     ROLLBACK TRANSACTION;

   --[Error logging section]
     DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
             @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
             @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE()
     EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;

   THROW; --will halt the batch or be caught by the caller's catch block
 END CATCH
END;
GO
```

After adding the trigger, we can test it. Let's try to add two new rows, each as single row. The first two inserts will work and data will be added. In the first insert, it will cause the MERGE statement to add a new row to the Example.AfterTriggerExampleGroupBalance table, the second updating that row:

```
INSERT INTO Example.AfterTriggerExample(AfterTriggerExampleId,GroupingValue,Value)
VALUES (1,'Group A',100);
GO
INSERT INTO Example.AfterTriggerExample(AfterTriggerExampleId,GroupingValue,Value)
VALUES (2,'Group A',-50);
```

Before we look at the summary table, let's check the case where the balance ends up being less than 0 with this row with -100 added the 100 and -50 from earlier causing a negative balance:

```
INSERT INTO Example.AfterTriggerExample(AfterTriggerExampleId,GroupingValue,Value)
VALUES (3,'Group A',-100);
```

This will cause the following error. Note that it uses the single row error message we set up, telling us the group that caused the error to help make error tracking easier:

www.it-ebooks.info

```
Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$InsertTrigger, Line 39
Grouping Value "Group A" balance value after operation must be greater than 0
```

Next, to show a multi-row error, I try to add two new rows, but not enough to make the sum greater than 0.

```
INSERT INTO Example.AfterTriggerExample(AfterTriggerExampleId,GroupingValue,Value)
VALUES  (3,'Group A',10),
        (4,'Group A',-100);
```

This causes the following error far more generic answer:

```
Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$InsertTrigger, Line 39
The total for the grouping value must be greater than 0
```

Of course, if you need better messages, you can clearly build a more interesting error handler, but since almost all code that may cause an error in most systems is going to be row at a time, it is probably not worth it. While it is fairly easy to build a single row error message the multi-row message is not going to be quite so easy to do, because you would need to know which rows were wrong, and that would make the cost of validation a lot more costly and a lot more coding.

For the error, that was raised, you will be able to see the error in the `utility.ErrorLog` table, if you implemented it:

```
SELECT *
FROM  utility.ErrorLog;
```

This will return at least the following row, depending on what other calls you have made:

```
ErrorLogId    Number    Location                              Message
----------    ------    --------                              -----------
22            50000     AfterTriggerExample$InsertTrigger     Grouping Value "Group A"
                                                              value after operation must be
                                                              greater than 0

LogTime                 ServerPrincipal
----------------------  --------------------
2012-02-21 00:06:27.35  DENALI-PC\AlienDrsql
```

Next, we will do another multi-row update that does not fail validation:

```
INSERT INTO Example.AfterTriggerExample(AfterTriggerExampleId,GroupingValue,Value)
VALUES (5,'Group A',100),
    (6,'Group B',200),
    (7,'Group B',150);
```

Now, let's look at the data that has been created:

```
SELECT *
FROM   Example.AfterTriggerExample;
SELECT *
FROM   Example.AfterTriggerExampleGroupBalance;
```


www.it-ebooks.info

This returns:

```
AfterTriggerExampleId   GroupingValue   Value
---------------------   -------------   -----
1                       Group A         100
2                       Group A         -50
5                       Group A         100
6                       Group B         200
7                       Group B         150

GroupingValue     Balance
-------------     -------
Group A           150
Group B           350
```

Take a look at the data, and make sure that the numbers add up. Test your code in as many ways as you possibly can. One of the reasons we don't generally write such messy code in triggers is that they will need a lot of testing. Also, all of the triggers that we are building basically maintain summary data that replaces optimizing a fairly simple query:

```
SELECT GroupingValue, SUM(Value) AS Balance
FROM   Example.AfterTriggerExample
GROUP  BY GroupingValue;
```

■ **Note** Sometimes it is useful/interesting to do an exercise like this to learn the difficulties in using code in certain ways to help you when you do need to do some complex code, and to show you why not to do this in your own code.

Next we move on to the UPDATE trigger. It is very similar in nature, and the validation section will be the same with the slight change to the FROM clause of the subquery to use a UNION of a query to the inserted and deleted table. You could make one validation trigger for INSERT, UPDATE, and DELETE, but I prefer to stick with one trigger with a bit of duplicated code, to make things easier to manage at the DBA level since multiple triggers complicates management.

Logically, an update is a delete and an insert of a new row. So when you change a row value, it deletes the old (represented in the deleted table), and creates a new row (in the inserted table). The MERGE statement for the UPDATE trigger has to deal with one additional branch to delete a group that has been added, so when you get a value, it could be a new group that was created by the update, a delete because the group from the deleted table no longer exists.

```
CREATE TRIGGER Example.AfterTriggerExample$UpdateTrigger
ON Example.AfterTriggerExample
AFTER UPDATE AS
BEGIN
  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  DECLARE @msg varchar(2000),  --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
```


www.it-ebooks.info

```
    @rowsAffected int = (SELECT COUNT(*) FROM inserted);
--   @rowsAffected int = (SELECT COUNT(*) FROM deleted);

--no need to continue on if no rows affected
IF @rowsAffected = 0 RETURN;

BEGIN TRY
   --[validation section]
   --Use a WHERE EXISTS to inserted to make sure not to duplicate rows in the set
   --if > 1 row is modified for the same grouping value
   IF EXISTS (SELECT AfterTriggerExample.GroupingValue
              FROM  Example.AfterTriggerExample
              --need to check total on any rows that were modified, even if key change
              WHERE EXISTS (SELECT *
                            FROM  Inserted
                            WHERE AfterTriggerExample.GroupingValue =
                                      Inserted.Groupingvalue
                            UNION ALL
                            SELECT *
                            FROM   Deleted
                            WHERE  AfterTriggerExample.GroupingValue =
                                       Deleted.Groupingvalue)
                            GROUP BY AfterTriggerExample.GroupingValue
                            HAVING SUM(Value) < 0)
              BEGIN
              IF @rowsAffected = 1
               SELECT @msg = CONCAT('Grouping Value "',
                 COALESCE(inserted.GroupingValue,deleted.GroupingValue),
                '" balance value after operation must be greater than 0')
               FROM  inserted --only one row could be returned…
                   CROSS JOIN deleted;
              ELSE
               SELECT @msg = CONCAT('The total for the grouping value must',
                         'be greater than 0');
               THROW 50000, @msg, 16;
              END

   --[modification section]
   --get the balance for any Grouping Values used in the DML statement
   SET ANSI_WARNINGS OFF; --we know we will be summing on a NULL, with no better way
   WITH GroupBalance AS
   (SELECT ChangedRows.GroupingValue, SUM(Value) AS NewBalance
    FROM  Example.AfterTriggerExample
    --the right outer join makes sure that we get all groups, even if no data
    --remains in the table for a set
         RIGHT OUTER JOIN
                    (SELECT GroupingValue
                     FROM Inserted
                     UNION
                     SELECT GroupingValue
                     FROM Deleted ) AS ChangedRows
         --the join make sure we only get rows for changed grouping values
```


www.it-ebooks.info

```
            ON ChangedRows.GroupingValue = AfterTriggerExample.GroupingValue
     GROUP BY ChangedRows.GroupingValue )
    --use merge because the user may change the grouping value, and
    --That could even cause a row in the balance table to need to be deleted
    MERGE Example.AfterTriggerExampleGroupBalance
    USING (SELECT GroupingValue, NewBalance FROM GroupBalance)
                  AS source (GroupingValue, NewBalance)
    ON (AfterTriggerExampleGroupBalance.GroupingValue = source.GroupingValue)
    WHEN MATCHED and Source.NewBalance IS NULL --should only happen with changed key
         THEN DELETE
    WHEN MATCHED THEN --normal case, where an amount was updated
         UPDATE SET Balance = source.NewBalance
         WHEN NOT MATCHED THEN --should only happen with changed
                               --key that didn't previously exist
            INSERT (GroupingValue, Balance)
            VALUES (Source.GroupingValue, Source.NewBalance);

    SET ANSI_WARNINGS ON; --restore proper setting, even if you don't need to
  END TRY
  BEGIN CATCH
   IF @@trancount > 0
     ROLLBACK TRANSACTION;

   --[Error logging section]
     DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
             @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
             @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE()
     EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;

   THROW; --will halt the batch or be caught by the caller's catch block

 END CATCH
END;
```

As a reminder, this is the where the balance is after the INSERT statement section:

```
GroupingValue    Balance
-------------    -------
Group A          150
Group B          350
```

So now, we update one row, setting a row that used to be 100 down to 50:

```
UPDATE Example.AfterTriggerExample
SET    Value = 50 --Was 100
WHERE  AfterTriggerExampleId = 5;
```

Looking at the balance again:

```
SELECT *
FROM   Example.AfterTriggerExampleGroupBalance;
```


www.it-ebooks.info

You will see that the balance has gone down to 100 for Group A.

```
GroupingValue     Balance
-------------     -------
Group A           100
Group B           350
```

The next thing to test is changing the value that is being grouped on. This will cause a row in the balance table to be deleted, and a new one to be added by the MERGE statement.

```
--Changing the key
UPDATE Example.AfterTriggerExample
SET    GroupingValue = 'Group C'
WHERE  GroupingValue = 'Group B';
```

Looking at the balance again:

```
GroupingValue    Balance
-------------    -------
Group A          100
Group C          350
```

You can see that the GroupValue has now changed from B to C, because the source data has all been changed. To set up the next example, let's change all of the rows to 10:

```
--all rows
UPDATE Example.AfterTriggerExample
SET    Value = 10 ;
```

This changes our data to look like:

```
SELECT *
FROM   Example.AfterTriggerExample;
SELECT *
FROM   Example.AfterTriggerExampleGroupBalance;
```

Returning:

```
AfterTriggerExampleId     GroupingValue     Value
---------------------     -------------     -----
1                         Group A           10
2                         Group A           10
5                         Group A           10
6                         Group C           10
7                         Group C           10

GroupingValue    Balance
-------------    -------
Group A          30
Group C          20
```


www.it-ebooks.info

Check to make sure a multi-statement failure works:

```
--violate business rules
UPDATE Example.AfterTriggerExample
SET    Value = -10;
```

This returns:

```
Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$UpdateTrigger, Line 45
The total for the grouping value must be greater than 0
```

Finally, we work on the DELETE trigger. It very much resembles the other triggers, except in this trigger we use the deleted table as our primary table, and we don't have to deal with the cases where new data is introduced, so we have one less matching criteria in the MERGE statement.

```
CREATE TRIGGER Example.AfterTriggerExample$DeleteTrigger
ON Example.AfterTriggerExample
AFTER DELETE AS
BEGIN
  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  DECLARE @msg varchar(2000),  --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
  --   @rowsAffected int = (SELECT COUNT(*) FROM inserted);
         @rowsAffected int = (SELECT COUNT(*) FROM deleted);

  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  BEGIN TRY
     --[validation section]
     --Use a WHERE EXISTS to inserted to make sure not to duplicate rows in the set
     --if > 1 row is modified for the same grouping value
     IF EXIStS (SELECT AfterTriggerExample.GroupingValue
                FROM  Example.AfterTriggerExample
                WHERE EXISTS (SELECT * --delete trigger only needs check deleted rows
                              FROM  Deleted
                              WHERE AfterTriggerExample.GroupingValue =
                                   Deleted.Groupingvalue)
                GROUP BY AfterTriggerExample.GroupingValue
                HAVING SUM(Value) < 0)
      BEGIN
        IF @rowsAffected = 1
                SELECT @msg = CONCAT('Grouping Value "', GroupingValue,
                       '" balance value after operation must be greater than 0')
                FROM  deleted; --use deleted for deleted trigger
        ELSE
         SELECT @msg = 'The total for the grouping value must be greater than 0';

        THROW 50000, @msg, 16;
     END
```


www.it-ebooks.info

```
      --[modification section]
      --get the balance for any Grouping Values used in the DML statement
       SET ANSI_WARNINGS OFF; --we know we will be summing on a NULL, with no better way
       WITH GroupBalance AS
       (SELECT ChangedRows.GroupingValue, SUM(Value) as NewBalance
        FROM  Example.AfterTriggerExample
        --the right outer join makes sure that we get all groups, even if no data
        --remains in the table for a set
             RIGHT OUTER JOIN
                  (SELECT GroupingValue
                   FROM Deleted ) AS ChangedRows
               --the join make sure we only get rows for changed grouping values
               ON ChangedRows.GroupingValue = AfterTriggerExample.GroupingValue
       GROUP BY ChangedRows.GroupingValue)

      --use merge because the delete may or may not remove the last row for a
      --group which could even cause a row in the balance table to need to be deleted
       MERGE Example.AfterTriggerExampleGroupBalance
       USING (SELECT GroupingValue, NewBalance FROM GroupBalance)
                                   AS source (GroupingValue, NewBalance)
       ON (AfterTriggerExampleGroupBalance.GroupingValue = source.GroupingValue)
       WHEN MATCHED and Source.NewBalance IS Null --you have deleted the last key
          THEN DELETE
       WHEN MATCHED THEN --there were still rows left after the delete
          UPDATE SET Balance = source.NewBalance;

      SET ANSI_WARNINGS ON; --restore proper setting
  END TRY
  BEGIN CATCH
   IF @@trancount > 0
     ROLLBACK TRANSACTION;

     --[Error logging section]
     DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
             @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
             @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE();
     EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;

    THROW; --will halt the batch or be caught by the caller's catch block

 END CATCH
END;
```

To test this code, I set up all of the balances to be 0 by setting several rows to -5 to balance out a 10, and one -10 to balance out the other 10.

```
UPDATE Example.AfterTriggerExample
SET    Value = -5
WHERE  AfterTriggerExampleId IN (2,5);

UPDATE Example.AfterTriggerExample
SET    Value = -10
WHERE  AfterTriggerExampleId = 6;
```


www.it-ebooks.info

This leaves the data in the following state:

```
AfterTriggerExampleId     GroupingValue     Value
----------------------    -------------     -----
1                         Group A           10
2                         Group A           -5
5                         Group A           -5
6                         Group C           -1
7                         Group C           10

GroupingValue     Balance
-------------     -------
Group A           0
Group C           0
```

First, we will try to delete the positive value from Group A:

```
DELETE FROM Example.AfterTriggerExample
WHERE  AfterTriggerExampleId = 1;
```

This gives you the following message:

```
Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$DeleteTrigger, Line 40
Grouping Value "Group A" balance value after operation must be greater than 0
```

Next, we will try deleting both of the positive values:

```
DELETE FROM Example.AfterTriggerExample
WHERE  AfterTriggerExampleId in (1,7);
```

This returns the generic multi-row error message that we created:

```
Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$DeleteTrigger, Line 38
The total for the grouping value must be greater than 0
```

Finally, we will systematically unload the table:

```
DELETE FROM Example.AfterTriggerExample
WHERE  AfterTriggerExampleId = 6;
```

Now you can see that the C group is 10 because we deleted the negative value:

```
GroupingValue   Balance
-------------   -------
Group A         0
Group C         10
```


www.it-ebooks.info

Now, we add back a Group B row:

```
INSERT INTO Example.AfterTriggerExample
VALUES (8, 'Group B',10);
```

Now our data looks like the following:

```
AfterTriggerExampleId   GroupingValue   Value
---------------------   -------------   -----
1                       Group A         10
2                       Group A         -5
5                       Group A         -5
7                       Group C         10
8                       Group B         10

GroupingValue   Balance
-------------   -------
Group A         0
Group B         10
Group C         10
```

Delete the entire Group A:

```
DELETE FROM Example.AfterTriggerExample
WHERE  AfterTriggerExampleId in (1,2,5);
```

Now the summary table looks like:

```
GroupingValue   Balance
-------------   -------
Group B         10
Group C         10
```

Finally, just clear the table:

```
DELETE FROM Example.AfterTriggerExample;
```

Leaving the tables both empty:

```
AfterTriggerExampleId   GroupingValue   Value
---------------------   -------------   -----

GroupingValue   Balance
-------------   -------
```

■ **Tip** All of this code to test the structures may seem like overkill but I definitely suggest that you do very much the same tasks on your own tables with triggers. I had to fix a lot of errors in my code as I worked through this example because in many cases, you have to deal with all of the particulars of inserts, updates, and deletes that a user can do.

www.it-ebooks.info

At this point, we have a set of triggers to maintain a summary table, but we haven't as yet covered the entire case. To deal with this completely you would need to add triggers to the `Example.AfterTriggerExampleGroupBalance` table to make sure that the row couldn't be modified unless it meets the criteria of summing up to values in the `Example.AfterTriggerExample` table or find a way to lock the table down completely, although any measures that prevent the dbo from changing the data would prevent the triggers we created from doing its job. I won't present that work here, but it is basically the same problem that is covered in Chapter 7 in the `AFTER` trigger section covering range checks on multiple rows.

INSTEAD OF Triggers

`INSTEAD OF` triggers are different from `AFTER` triggers in that they fire prior to the DML action being affected by the SQL engine. In fact, when you have an `INSTEAD OF` trigger on a table, it's the first thing that's done when you `INSERT`, `UPDATE`, or `DELETE` from a table. These triggers are named `INSTEAD OF` because they fire *instead of* the native action the user executed. Inside the trigger, you perform the action manually—either the action that the user performed or some other action.

Instead of triggers have a fairly narrow set of use cases. The most typical use is to automatically populate a value such as the last time a change to a row occurred (and since you perform the action in your code, no DML operation can override your action.

A second value of instead of triggers is that they can be created on a view to make a view editable in a straightforward manner for the client. Why some views are editable, you are only allowed to update a single table at a time. By applying an instead of trigger to the view, you control the DML operation and you can insert to multiple tables in the background. Doing this, you encapsulate calls to all the affected tables in the trigger, much like you would a stored procedure, except now this view has all the properties of a physical table, hiding the actual implementation from users.

Probably the most obvious limitation of `INSTEAD OF` triggers is that you can have only one for each action (`INSERT`, `UPDATE`, and `DELETE`) on the table, or you can combine them just as you can for `AFTER` triggers, which I strongly advise against for `INSTEAD OF` triggers. We'll use pretty much the same trigger template that we used for the T-SQL `AFTER` triggers, with only the modification that now you have to add a step to perform the action that the user was trying to do, which I comment as `<perform action>`. This tag indicates where we will put the DML operations to make modifications.

```
CREATE TRIGGER <schema>.<tablename>$InsteadOf<actions>Trigger
ON <schema>.<tablename>
INSTEAD OF <comma delimited actions> AS
BEGIN
  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  DECLARE @msg varchar(2000),  --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
      @rowsAffected = (SELECT COUNT(*) FROM inserted);
  --@rowsAffected = (SELECT COUNT(*) FROM deleted);

  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  BEGIN TRY
     --[validation section]
     --[modification section]
     --<perform action>
```


www.it-ebooks.info

```
  END TRY
  BEGIN CATCH
   IF @@trancount > 0
       ROLLBACK TRANSACTION

   --[Error logging section]
   DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
           @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
           @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE();
   EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;

   THROW; --will halt the batch or be caught by the caller's catch block

 END CATCH
END;
```

The most difficult part of the INSTEAD OF trigger is that you have to perform the operation yourself, meaning you have to maintain the triggers anytime the table needs to change. Technically, performing the action is optional, and in the examples in Chapter 7, I use INSTEAD OF triggers to prevent a DML operation from occurring altogether.

I most often use INSTEAD OF triggers to set or modify values in my statements automatically so that the values are set to what I want, no matter what the client sends in a statement. A good example is a column to record the last time the row was modified. If you record last update times through client calls, it can be problematic if one of the client's clock is a minute, a day, or even a year off. (You see this all the time in applications. My favorite example was in one system where phone calls appeared to be taking negative amounts of time because the client was reporting when something started and the server was recording when it stopped.) It's generally a best practice not to use INSTEAD OF triggers to do validations and to use them only to shape the way the data is seen by the time it's stored in the DBMS. In Chapter 7, I demonstrated four ways you can use INSTEAD OF triggers:

- *Automatically maintained columns*: Automatically setting a value, like the point in time when a row was last updated
- *Formatting user input*: Forcing a value in a table to meet a given format, like forcing values in a column to be all CAPS
- *Redirecting invalid data to an exception table*: Taking values that are outside of a given tolerance, and instead of returning an error, pushing the error off to be looked at later
- *Forcing no action*: Stopping a DML action to be performed on a table, even by someone who technically has proper rights

---

■ **Note** INSTEAD OF triggers tend to really annoy some developers that want complete control over the changing of data, even more than an AFTER trigger that usually does something that the non-data layer might not be able to do. I am not asserting that triggers are always the best way, though I do prefer having data tier level control over some functions so that certain operations (such as an capturing who modified a row, and when it was modified) can be guaranteed to occur, no matter how many ways the developer ends up affecting changes to the table.

I will do an example very similar to the automatically maintained columns example. The table follows, with an integer key, a column that will be formatted, and a couple of columns included in some tables to tell when they were last modified.

www.it-ebooks.info

```
CREATE TABLE Example.InsteadOfTriggerExample
(
    InsteadOfTriggerExampleId int NOT NULL
            CONSTRAINT PKInsteadOfTriggerExample PRIMARY KEY,
    FormatUpper varchar(30) NOT NULL,
    RowCreatedTime datetime2(3) NOT NULL,
    RowLastModifyTime datetime2(3) NOT NULL
);
```

This example is a lot simpler than the AFTER trigger example, simply replacing the insert statement with one that does a bit of formatting on the incoming data. Generally speaking, the operations you will need to do in INSTEAD OF triggers are going to be very simple and straightforward.

```
CREATE TRIGGER Example.InsteadOfTriggerExample$InsteadOfInsertTrigger
ON Example.InsteadOfTriggerExample
INSTEAD OF INSERT AS
BEGIN
  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  DECLARE @msg varchar(2000),  --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
       @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  --   @rowsAffected int = (SELECT COUNT(*) FROM deleted);

  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  BEGIN TRY
        --[validation section]
        --[modification section]
        --<perform action> --this is all I change other than the name and table in the
                           --trigger declaration/heading
        INSERT INTO Example.InsteadOfTriggerExample
                    (InsteadOfTriggerExampleId,FormatUpper,
                     RowCreatedTime,RowLastModifyTime)
        --uppercase the FormatUpper column, set the %time columns to system time
         SELECT InsteadOfTriggerExampleId, UPPER(FormatUpper),
                SYSDATETIME(),SYSDATETIME()
         FROM  inserted;
  END TRY
  BEGIN CATCH
   IF @@trancount > 0
       ROLLBACK TRANSACTION;

   --[Error logging section]
   DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
           @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
           @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE();
   EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;
```


www.it-ebooks.info

```
   THROW; --will halt the batch or be caught by the caller's catch block
 END CATCH
END;
```

Start out with a simple insert, referencing only two of the columns and not the row modification columns:

```
INSERT INTO Example.InsteadOfTriggerExample (InsteadOfTriggerExampleId,FormatUpper)
VALUES (1,'not upper at all');
```

Now, checking the data:

```
SELECT *
FROM   Example.InsteadOfTriggerExample;
```

You can see that the FormatUpper value has been set to all uppercase, and the RowCreatedTime and RowLastModify time values have been set:

```
InsteadOfTriggerExampleId   FormatUpper         RowCreatedTime           RowLastModifyTime
-------------------------   ----------------    ----------------------   ----------------------
1                           NOT UPPER AT ALL    2012-05-22 21:47:50.56   2012-05-22 21:47:50.56
```

Now add two rows at a time to make sure that multi-row operations work:

```
INSERT INTO Example.InsteadOfTriggerExample (InsteadOfTriggerExampleId,FormatUpper)
VALUES (2,'UPPER TO START'),(3,'UpPeRmOsT tOo!');
```

This will also be formatted as expected. Check the data:

```
InsteadOfTriggerExampleId   FormatUpper        RowCreatedTime           RowLastModifyTime
-------------------------   ----------------   ----------------------   ----------------------
1                           NOT UPPER AT ALL   2012-05-22 21:47:50.56   2012-05-22 21:47:50.56
2                           UPPER TO START     2012-05-22 21:48:18.28   2012-05-22 21:48:18.28
3                           UPPERMOST TOO!     2012-05-22 21:48:18.28   2012-05-22 21:48:18.28
```

Now, check the error handler. Unlike AFTER triggers, you shouldn't really expect any errors since almost any error you might check with an instead of trigger would be better served in a constraint in almost all cases. However, when you have an instead of trigger, you do get the constraint errors being trapped by the trigger (thankfully, with the error re-throwing capabilities of THROW in 2012 the error will be the native error that occurs):

```
--causes an error
INSERT INTO Example.InsteadOfTriggerExample (InsteadOfTriggerExampleId,FormatUpper)
VALUES (4,NULL) ;
```

This returns the following error, which you can see claims to be coming from the InsteadOfTriggerExample $InsteadOfTrigger, line 23 in case you need to debug:

```
Msg 515, Level 16, State 2, Procedure InsteadOfTriggerExample$InsteadOfInsertTrigger, Line 23
Cannot insert the value NULL into column 'FormatUpper', table 'tempdb.Example.
InsteadOfTriggerExample'; column does not allow nulls. INSERT fails.
```


www.it-ebooks.info

Finally, we write the UPDATE version of the trigger. In this case, the UPDATE statement again forces the values of the RowLastModifyTime column to ignore whatever is passed in, and ensures that the RowCreatedTime never changes.

```
CREATE TRIGGER Example.InsteadOfTriggerExample$InsteadOfUpdateTrigger
ON Example.InsteadOfTriggerExample
INSTEAD OF UPDATE AS
BEGIN
  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount
  DECLARE @msg varchar(2000),  --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
       @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  --   @rowsAffected int = (SELECT COUNT(*) FROM deleted);

  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;

  BEGIN TRY
     --[validation section]
     --[modification section]
     --<perform action>
     --note, this trigger assumes non-editable keys. Consider adding a surrogate key
     --(even non-pk) if you need to be able to modify key values
     UPDATE InsteadOfTriggerExample
     SET  FormatUpper = UPPER(inserted.FormatUpper),
         --RowCreatedTime, Leave this value out to make sure it was updated
         RowLastModifyTime = SYSDATETIME()
     FROM  inserted
          JOIN Example.InsteadOfTriggerExample
            ON inserted.InsteadOfTriggerExampleId =
                 InsteadOfTriggerExample.InsteadOfTriggerExampleId;
  END TRY
  BEGIN CATCH
   IF @@trancount > 0
       ROLLBACK TRANSACTION;

   --[Error logging section]
   DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
           @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
           @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE();
   EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;

   THROW;--will halt the batch or be caught by the caller's catch block

 END CATCH
END;
```

Simple, really; just formatting data. Now update our data, two rows set to final test, and not that I set the modify times in the UPDATE statement, though they won't be honored in the final results:

```
UPDATE Example.InsteadOfTriggerExample
SET    RowCreatedTime = '1900-01-01',
       RowLastModifyTime = '1900-01-01',
```


www.it-ebooks.info

```
       FormatUpper = 'final test'
WHERE  InsteadOfTriggerExampleId in (1,2);
```

Now check the data:

```
InsteadOfTriggerExampleId    FormatUpper       RowCreatedTime           RowLastModifyTime
-------------------------    -------------     ----------------------   ----------------------
1                            FINAL TEST        2012-05-22 21:47:50.56   2012-05-22 21:50:23.70
2                            FINAL TEST        2012-05-22 21:48:18.28   2012-05-22 21:50:23.70
3                            UPPERMOST TOO!    2012-05-22 21:48:18.28   2012-05-22 21:48:18.28
```

The row created times are the same as they were, and the modify times for the two rows are set to the time when the rows were updated (at the time I wrote this example).

It's important to note that if you use a column with the identity property for a surrogate key, using an instead of trigger makes the SCOPE_IDENTITY() function cease to work because the modification statement is not in the same scope. For example, take the following small table:

```
CREATE TABLE testIdentity
(
       testIdentityId int IDENTITY CONSTRAINT PKtestIdentity PRIMARY KEY,
       value varchar(30) CONSTRAINT AKtestIdentity UNIQUE,
);
```

Without an instead of trigger, you can do the following:

```
INSERT INTO testIdentity(value)
VALUES ('without trigger');
SELECT SCOPE_IDENTITY() AS scopeIdentity;
```

And this will return:

```
scopeIdentity
-------------
1
```

But add a trigger such as the following (which does nothing but insert the data as-is, for the example):

```
CREATE TRIGGER testIdentity$InsteadOfInsertTrigger
ON testIdentity
INSTEAD OF INSERT AS
BEGIN
  SET NOCOUNT ON; --to avoid the rowcount messages
  SET ROWCOUNT 0; --in case the client has modified the rowcount

  DECLARE @msg varchar(2000),  --used to hold the error message
  --use inserted for insert or update trigger, deleted for update or delete trigger
  --count instead of @@rowcount due to merge behavior that sets @@rowcount to a number
  --that is equal to number of merged rows, not rows being checked in trigger
       @rowsAffected int = (SELECT COUNT(*) FROM inserted);
  --   @rowsAffected int = (SELECT COUNT(*) FROM deleted);

  --no need to continue on if no rows affected
  IF @rowsAffected = 0 RETURN;
```


www.it-ebooks.info

```
  BEGIN TRY
        --[validation section]
        --[modification section]
        --<perform action>
        INSERT INTO testIdentity(value)
        SELECT value
        FROM  inserted;
  END TRY
  BEGIN CATCH
   IF   @@trancount > 0
        ROLLBACK TRANSACTION;

   --[Error logging section]
   DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
           @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
           @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE();
   EXEC Utility.ErrorLog$Insert @ERROR_NUMBER,@ERROR_PROCEDURE,@ERROR_MESSAGE;

   THROW ;--will halt the batch or be caught by the caller's catch block

 END CATCH
END;
```

And you will see that running very similar code:

```
INSERT INTO testIdentity(value)
VALUES ('with trigger');

SELECT SCOPE_IDENTITY() AS scopeIdentity;
```

Results in a NULL value:

```
scopeIdentity
-------------
NULL
```

My typical solution is to use my knowledge of the data structures to simply use the natural key:

```
INSERT INTO testIdentity(value)
VALUES ('with trigger two');

SELECT testIdentityId AS scopeIdentity
FROM   testIdentity
WHERE  value = 'with trigger two'; --use an alternate key
```

And this returns:

```
scopeIdentity
-------------
3
```

Or, even using a sequence-based key (see Chapter 6), which can be defaulted, or the caller can fetch the next value and pass it into the insert. In any case, the problem is often that the tool the developer uses expects the SCOPE_IDENTITY function to work, so it can obviate the ability to use an instead of insert trigger.

www.it-ebooks.info

Index

A

AccessKey, 171, 177
AccessKeyValue, 196
AccountName: name1, 134
Ad hoc SQL, 596
  advantages and disadvantages, 597
  parameterization
    ALTER DATABASE command, 605
    CONVERT_IMPLICIT(nvarchar(4000),[@0],0) command, 605
    encapsulation layer, 607
    LIKE condition, 605–606
    N'1, rue Pierre-Demoulin' value, 604–605
    simple query, 604
    sp_executeSQL, 606–607
    sp_unprepare and sp_prepare statement, 607
    SqlCommand object, 606
  pitfalls
    batching interface, 610
    code-wrapping mechanism, 609
    functional code, transaction, 609
    low Cohesion and high coupling, 607–608
    performance tuning, 612–613
    security issues, 610–611
    SQL injection, 611–612
  runtime control over queries
    bolded query, 600–601
    FROM clause, 598
    contact table, 599–600
    database creation, 598–599
    data changed property, 601
    IF blocks, 603
    sales.contact table, 601
    sales summary column, 600–601
    SELECT clause, 598
    UPDATE statements, 601
    varchar(max) columns, 601
    WHERE clause, 602
    Windows file-listing dialog, 599
  shared execution plans, 603–604
  SQL Server Management Studio, 597
AFTER triggers, 261, 711
  audit trail maintenance
    change tracking and data capture, 277
    columnname, 279
    employee table, 277
    tablename, 279
  cascading inserts, 270–273
  child to parent cascading, 273–276
  databases and servers relationship
    child insert and child update, 281–283
    demographics database, 279
    parent delete, 281
    parent update, 280
  DELETE trigger, 278, 723–726
  disallow results set, 714
  Error logging section, 712
  Example.AfterTriggerExample, 715
  Example.AfterTriggerExampleGrouping Balance, 715–716
  GroupValue, 715, 722
  INSERT statement, 721
  MERGE operation and statement, 714, 717, 722
  modification section, 712
  multiple rows
    Accounting.AccountActivity table, 262–263
    accounting groups schema, 262
    AccountNumber, 264
    account table, 262
    FROM clause, 264
    error and updation, 718–719
    error message, 264–265
    trigger events, 265
    WHERE clause, 264
  summary value maintenance

www.it-ebooks.info

AFTER triggers (*cont.*)
AccountActivity and Account query, 269
AccountActivity rows, 266, 268
Account table, 268
BalanceAmount column addition and updation, 266
Balance column updation, 266
DELETE trigger, 269–270
EXISTS filter, 266–268
nested triggers, 270
recursive triggers, 270
trigger creation, 711–712
TriggerDemo database, 713
types, 261
UPDATE trigger, 278, 719–721
utility.ErrorLog table, 718
validation and cascading/modification action, 716–717
validation section, 712
Aggregation reporting style, 641
additional summary tables, 667
benefits, summary modeling, 664
initial summary table, 665–666
AllowMarketingByEmailFlag column, 155
Alternate key (AK), 19
Analytical reporting style
Bill Inmon approach, 640
denormalization, 640
dimensional modeling
adjudication process, 644
business process, 642, 644
date dimension, 644–647
health care payer dimensional model, 642–643
slowly changing dimension, 647–652
snapshot fact, 659–661
snowflake dimension, 652–653
transaction fact(*see* Transaction fact)
type dimension, 653–655
Ralph Kimball's approach, 640
Anti-patterns, 301
generic key references, 359
GUID key, 365
JournalEntry, 364–366
multiple tables, same key, 364
objects, maximum usability/flexibility, 366
RelatedTableName, 365
SalesOrders, 364
TroubleTickets, 364
invisible physical layer, 359
no datatype standardization, 360
one-size-fits-all key domain, 359
cached object, 363
Customer table, 361
data in query, 362
domain tables, 362
domain values, 361
expandability and control, 363
foreign key constraints, 363
normalization process, 362
one domain table per purpose, 363
one multiuse domain table, 362
one table application, 364
performance considerations, 363
relational databases, 361
overusing unstructured data, 359, 367–368
poor domain choices, 360
poor normalization practices, 360
undecipherable data, 359, 360–361
Approximate uniqueness, 318–319
AttendeeNumber, 171
Attendees.AttendeeType, 227
Attendees.MessagingUser table, 218
AttendeeType, 177, 229
columns, 196
domain, 186–188
table, 201
Auto dealer submodel, 154
automobileMake row, 178
automobileModelStyle table, 178
AutomobileSale table, 155

B

bCarFlag, 176
BCNF. *See* Boyce-Codd Normal Form
bigint datatype, 188
Bill Inmon approach, 640
binary(N) datatype, 189
bIsCar, 176
bit datatype, 188, 191
BookISBN key, 145
Boolean/logical values, 191–192
Boyce-Codd Normal Form (BCNF), 144
multiple columns with same prefix, 153
repeating groups of data, 154
summary data, 154–155
BuildInstance, 312
Bulk uniqueness
baseplate types, 311
bricks types, 311
BuildInstance, 308, 311
BuildInstancePiece, 311
CTE, 312
data loading, 309
grouping sets, 311–312
InventoryAdjustment Quantity, 307
InventoryAdjustment table, 306
inventory and utilization storage, 306

www.it-ebooks.info

inventory model, 307
Lego® collection, 307
minimal builds, 310
personal inventory, 308
Piece table, 311–313
ProductSale Quantity, 307
row constructor syntax, 310
rows and data manipulation, 306
sample Lego parts, 307
table creation, 308
table implementation, 309
types of pieces, 309–310
BusinessEntityID, 570

C

Calendar table, 564, 593
BigSaleDaysFlag column, 579
Business Intelligence/OLAP implementations, 576
ON clause, 584
CTEs, 578–579
datename and datepart functions, 576
final table, 580–582
fiscal calendar, 585
fiscal time periods, 580
FORMAT command, 576
insert statement, 585
natural relational coding style/technique, 578
normalization and denormalization, 576
OLTP databases, 576
OrderDate OrderTime, computed column, 578
payroll system, 585
query, fiscalYear, 582
relative week count, 583–584
2012 RTM version, 584
table creation, 576
time, floating windows, 580–582
year column, 577
CallId, 282
CampActivity, 341
Candidate key
Book_Name, 20
composite key, 18
definition, 18
ISBN_number, 20
natural keys, 20–22
Publisher_Name, 20
Smokey, 19
surrogate keys, 23–24
types, 19, 20
char(N) datatype, 189
CHARINDEX function, 192
Check constraints
ALTER TABLE statement, 250, 252
catalog number mask, 249
WITH CHECK, 251
chkMusic_Artist$Name$noMadonna Names, 249, 250, 252
chkMusic_Artist$Name$NoPetShopNames, 249
complex expressions, 248
data seeding, 249
declarative constraints, 247
error handling, 247, 258
errors enhancement
chkMusicAlbum$CatalogNumber$CatalogNumberValidate, 258, 261
entering invalid value, 260
error description, 258
ErrorMap table, 259
error messages, 258, 260
error number, 258
level, 258
line, 258
mapping table, 258–259
rudimentary error-mapping scheme, 258
state, 258
TRY-CATCH error handling, 258
example schema, 248
INSERT statement, 249
WITH NOCHECK setting, 250
release-date column, 249
simple expressions, 248
chkMusicAlbum$Name$noEmptyString, 253
date range checks, 252–253
empty strings, 252
LEN function, 253
SSIS OLEDB output, 254
trusted status and values, 254
value reasonableness, 253
sys.check_constraints catalog object, 251
sys.check_constraints query, 252
tables creating and populating code, 248–249
triggers errors
constraint mapping function, 297
doomed transaction, 296
ROLLBACK, 293
SELECT statement, 294
THROW, 297
transaction state, 294
transaction without rolling back, 295–296
TRY-CATCH block, 292
utility.ErrorLog$insert object, 297
UPDATE statement, 250
using functions
Album table, 257
Boolean expression, 254, 255
CatalogEntryMask, 257
CatalogNumber column, 255–257

www.it-ebooks.info

CHARINDEX function (*cont.*)
CatalogNumberMask column, 256
CLR-based objects, 255
complex scalar validations, 255
DML modification statement, 255
Publisher.CatalogNumberMask, 255
Publisher table, 257
query, 256
row with invalid value, 256
single-row validation code, 257
tables access validation, 255
T-SQL function, 255–256
UDFs, 255, 257
Chen Entity Relationship Model (ERD) methodology, 85–86
ChildName, 151
chunked updates, 193
CLR, 599, 610
complex procedural logic and computations, 629
extended stored procedures, 630
guidelines, 605
.NET framework classes, 630
objects and functionality, 630
object types
stored procedures, 635
triggers, 635
user-defined aggregates, 636
user-defined functions, 634–635
user-defined types, 636–637
rich language support, 629
string manipulation, complex statistical calculations and custom encryption, 630
Visual Studio, 630
Codd's eighth rule, 446
Code-based denormalizations, 207
colFirstName, 176
Collation, 199–200
columnCity, 176
Comma delimited items, 571–573
Common Table Expression (CTE), 312
Concurrency coding, 560–561
isolating sessions
isolation levels (*see* Isolation levels)
lock characteristics, 528
lock modes, 530–532
lock types, 529–530
lost update, 528
logical unit, 558–560
MARS, 507
multiuser database, 507
optimistic locking, 552–553
OS and hardware, 508–509
parallelism, 507
pessimistic locking, 546–548
Resource Governor, 505–506
row-based locking
add a time column method, 553
check all columns method, 553
optimistic lock columns, 553–556
row-level optimistic lock coding, 557–558
timestamp column method, 553
single-threaded code block, 549–551
tradeoffs, 506
transactions
ACID, 509
basic transactions, 510–513
bulk logged model, 510
checkpoint, 509
definition, 509
distributed transactions, 510, 517–518
DML and DDL statement, 510
explicit *vs.* implicit transactions, 518
full model, 510
MARS, 517
multiple commands, 510
nesting transactions, 510, 513–515
savepoints, 510, 515–516
simple model, 510
SQL Server Code (*see* SQL Server Code)
T-SQL code, 510
ConnectedToUser, 171
contact$delete procedure, 619
Contacts.Journal table, 282
CREATE SEQUENCE dbo.test, 211
CREATE TABLE statements, 204–205
CTE. *See* Common Table Expression
CustomerId, 156

D

Data access strategies, 595, 638
ad hoc SQL (*see* Ad hoc SQL)
Entity Framework, 596
interface creation, 596
non-data tier rule, 596
object-relational mapping tools, 596
security and indexing strategies, 595
stored procedures, 637
advantages and disadvantages, 628
AdventureWorks2012 table, 613–614
complex plan parameterization, 621–622
cross-platform coding, 623
data format, 628
data-manipulation code, 628
DBA, 629
dbo powers/sysusers, 617
encapsulation, 614–616
EXECUTE AS clause, 619

www.it-ebooks.info

fine-tuning without program changes, 622–623
high initial effort, 623
injection attack value, 616–617
INSTEAD OF trigger, 626–627
JOIN clause, 618
mutable business logic and rules, 628
optional parameters, 623–624
precompiled stored procedures, 616
quotename() function, 617
RETURN statement, 613
sales.contact table, 601–602
salesLevelId column, 624
salesLevel value, 625
security, 614, 619–621
securityEXECUTE AS clause, 619–620
sp_executeSQL and parameterization, 618
structure, 613
WHERE clause, 618
Database access prerequisites
contained database model, 376
advantages, 379
ALTER DATABASE statement, 380
authentication, 380
ContainedDbSecurityExample database, 383
contained user, 380
CREATE DATABASE statement, 380
CREATE USER statement, 380
logging demonstration, 381
Object Explorer, 382
sp_configure, 380
virtualization, 379
WilmaContainedUser, 380
database principals, 373
database scope securables, 374
DENY security statement, 374, 375
GRANT security statement, 374, 375
impersonation, 383–386
login and user
CHECK_POLICY setting, 376
ClassicSecurityExample database, 378
CREATE LOGIN statement, 376
database principal, 379
FRED user rights, 377
Management Studio user, 377
password, 376
REVOKE statement, 378
server permissions, 377
server principal, 379
SQL Server authentication, 377
tempdb database, 376
Windows Authentication, 376
REVOKE security statement, 374, 375
schema scope securables, 374
server-scoped securables, 374
server security guidelines, 373–374
SQL Server principals, 373
windows principals, 374
Database-level security
impersonation, 404–409
stored procedures and scalar functions, 402–404
views and table-valued functions
column-level security, 411
data, projection/vertical partitioning, 411
error reporting, 413
GRANT syntax, 413
multistatement table-valued function, 411
ProductViewer group, 412
row-level security, 413–417
Database lines
certificate-based trust, 424–426
cross-database chaining
ALTER AUTHORIZATION statements, 420
ALTER AUTHROIZATION DDL statement, 419
ALTER DATABASE, 419
containment level, 421
cross database access, 419
Cross DB Ownership Chaining, 419
DB_CHAINING database option, 418
externalDb database, 420
OLTP databases, 418
SELECT statement, 421
server principal, 421
smurf principal, 420, 421
sys.databases catalog view, 419, 420
TRUSTWORTHY database option, 418
database-backup scenario, 417
distributed queries, 426–427
foreign key constraints, 417
impersonation, 423–424
Database securables
application roles, 397–400
built-in database roles, 392–393
column-level security, 390–392
grantable permissions, 387–388
object groupings, 388
schemas, 400–402
table security, 388–390
user-defined database roles
Administrators role, 393
database-level rights, 393
DENY operation, 396
Employees role, 394
GRANT operations, 396
HRManagers role, 393, 396
HR system, 395
HRWorkers role, 393, 396
human resources system, 393
Managers role, 394

www.it-ebooks.info

Database securables (*cont.*)
Payroll schema, 395, 396
T-SQL code, 394
users security information, 394
Data definition language (DDL)
basic check constraints
Boolean expression, 224, 226
MessageTopic table, 226
NULL, 224
TopicName, 225
triggers, maintain automatic values, 226–229
T-SQL, 224
user-defined functions, 224
UserHandle, 225
validation routines, 224
[WITH CHECK | WITH NOCHECK] specification, 225
basic table structures
columns and base datatypes, 206–208
nonnatural primary keys (*See* Nonnatural Primary Keys)
nullability, 208
schema, 205–206
ConferenceMessaging database, 202–203
database design/generation tool, 202
database documentation
Description script, 231–232
Descriptions in Management Studio, 232
fn_listExtendedProperty object, 232–233
functions and procedures, 230
Messages.Topic table, 231
parameters, 230
reindexing schemes, 231
repository information, 230
tables, column, and schemas limitations, 233
data modeling tools, 202
default constraints, 218–219
Development Studio, 202
foreign keys
addition syntax, 220
ALTER TABLE statement, 219
AttendeeType, 221
CASCADE operations, 222
CREATE TABLE statement, 220
cross-database relationships, 224
messaging model, 221
MessagingUser and Message, 222, 223
parent-to-child relationship, 220
SQL Azure, 224
surrogate keys, 220–221
Topic and MessageTopic, 223
UserConnection, 222
verb phrases, 221
metadata
catalog view, 237
INFORMATION_SCHEMA views, 234, 237
schema list, 233
sys schema objects, 233
table and column name, 234–237
uniqueness constraints addition
alternate key constraints, 216–217
indexes, 214, 217–218
primary key constraints, 214–216
Data gathering, 37
best practices, 50
client interviews
communication gap, 42
formal, 42
one-to-one sessions, 43
structured database, 42
data integration with system, 47
data location, 46
data type needed and used, 43–44
data worth, 47
documentation
audit plans, 50
contracts/client work order, 49
early project, 49
service level agreement, 49
governing rules, 44–45
multiuser data, 47
reported data, 45
requirements, 38
artifacts, 42
document, 39–40
E-R modeling method, 41
graphical model, 42
large system, 41
Scrum methodology, 41
UML, 41
systems and prototypes, 48
waterfall process, 38
Data Manipulation Language (DML)
triggers, 434–438, 707
advantages, 261
AFTER triggers, 261, 721, 726
audit trail maintenance, 276–279
cascading inserts, 270–273
child to parent cascading, 273–276
databases and servers relationship, 279–283
DELETE trigger, 723–726
disallow results set, 714
Error logging section, 712
Example.AfterTriggerExample, 715
Example.AfterTriggerExampleGroupingBalance, 715–716
GroupValue, 715, 722

www.it-ebooks.info

INSERT statement, 721
MERGE operation, 714
MERGE statement, 717, 722
modification section, 712
multi-row error, 718
multi-row updation, 718–719
range checks, multiple rows, 262–265
summary value maintenance, 265–270
trigger creation, 711–712
TriggerDemo database, 713
types, 294
UPDATE trigger, 719–721
utility.ErrorLog table, 718
validation and cascading/modification action, 716–717
validation section, 712
and constraints errors
constraint mapping function, 297
doomed transaction, 296
ROLLBACK, 293
SELECT statement, 294
THROW, 297
transaction state, 294
transaction without rolling back, 295–296
TRY-CATCH block, 292
utility.ErrorLog$insert object, 297
database option–ALTER DATABASE–RECURSIVE_TRIGGERS, 709
deleted table, 707
disallow results from triggers option, 709
error logging procedure, 711
EXISTS Boolean expression, 709
inserted table, 707
INSTEAD OF triggers, 261
applications, 728
DBMS, 284
error handler, 730
formatting user input, 284–287
FormatUpper value, 730
insert statement, 729–730
multi-row operations, 730
no action on table, 290–292
<perform action> comment, 727–728
redirecting invalid data, 287–290
RowCreatedTime and RowLastModify time values, 730–731
SCOPE_IDENTITY() function, 732–733
UPDATE statement, 731
modified columns determination, 708
multiple AFTER triggers, 708
multi-row considerations, 708
nesting triggers, 708
OLTP operations, 260
performance, 708
recommendation, 260–261
SELECT statement, 709
server option–sp_serveroption–nested triggers, 709
stored procedure, 707
trigger usage, 710
T-SQL command, 710
types, 707
Utility.ErrorLog, 711
Data modeling, 71
alternate methodology
Chen ERD, 85–86
database diagramming, 87
information engineering (IE), 83–85
Internet, 83
Visio, 86
attributes
alternate key, 61–62
building models, 7
Customer and Invoice entity, 117
descriptive information, 114
descriptive metadata, 118–119
domain (*see* Domains)
final graphical model, 120
foreign key, 62–63
graphical model, 117
identification, information, 111
identifiers (*See* Identifiers)
list of names, 7
locators, 114–116
naming, 65–68
primary key, 59–61
transformed, normalization process, 59
values, 116
best practices, 88, 127
business rules, 121–122
client management, 93
conceptual model, 92, 109
database design, 54
descriptive information
basic set, 82
documentation process, 81
maintenance, 82
scope, 82
storage, 81
documentation, 93
EmployeeRelationship, 110
entity
abbreviations, 57
audit trails, 99
dependent, 56
document, 97–98
events, 99
foreign key, 55
group, 98

www.it-ebooks.info

Data modeling (*cont.*)
Hungarian notation, 57
ideas, 96–97
identification, 93
independent, 56
invoiceNumber, 55
list of preliminary entities, 100–101
logical naming, 57
naming database objects, 56
objects, 95–96
Pascal/mixed case, 57
people, 94
places, 95
primary key, 55
records and journals, 100
singular/plural names, 57
fundamental process, 123–124
graphical nature, 54
IDEFIX, 53–55
initial relationship
documentation, 108–109
logical model, 92
additional data needed, 125, 126
analysis paralysis, 124
client review, 126
customer agreement, 126
many-to-many relationship, 106–107
objectives, 91
one-to-N relationship
association, 102–104
Is-A relationship, 105–106
multivalued attribute, 103, 105
relational model, 102
transaction, 103–105
relationship
cardinality, 72–74
graphical representation, 67
identification, 68
many-to-many, 78–79
migrated key, 67
nonidentification, 69–71
recursive, 74–76
role name, 71–72
subtypes, 76–78
types, 67, 68
verb phrases, 79–81
requirements, 93
Data obfuscating, 427–429
Data protection, 245
check constraints (*see* Check constraints)
concurrency, 246
custom front-end tools, 246
database-engine level, 246
data formatting, 245
DML triggers (*see* Data Manipulation Language triggers)
generic data manipulation tools, 246
nvarchar(20) column, 246
raw queries, 246
reliability and integrity, 245
routines, 246
UNIQUE constraint, 245
Date datatype, 189
Date dimension
entity representation, 645
IDENTITY column, 645
standard attributes, 645
stored procedure, 646
table creation, 645
unknown rows, 646
datetime, 189
datetime2(N) datatype, 189
datetimeoffset, 189
dbo.doAnything procedure, 621
dbo schema, 205
DDL. *See* Data definition language
DDL triggers, 438–441
decimal datatype, 189
Delimited identifiers, 173
Denormalization, 152, 162–165
Dependency
entire key, 147–149
independent multivalue, 157–159
partial key, 146–147
between rows, 152
surrogate keys effect
DoorCount, 149
driversLicense and employee tables with natural keys, 149
driversLicense and employee tables with surrogate keys, 150
driversLicense table, 151
GrandparentId, 151
Grandparent table, 151
key migration, 151
Parent table, 151
StateCode, 150
VehicleTypeId, 149
DesiredModelLevel, 146
Development Studio, 202
DisabledFlag, 218
DistributionDescription, 175
Domain: domain1.com, 134
Domain implementation
collation selection, 199–200
column/table, 186–189
consistency, 184
datatype selection

www.it-ebooks.info

approximate numeric data, 189
binary data, 189
boolean/logical values, 191–192
character (or string) data, 189
complex datatypes, 194–195
date and time, 189
deprecated/bad choice types, 190–191
heirarchyId, 195
large-value datatype columns, 192–193
messaging system model, 197
MessagingUser table, 195–196
precise numeric data, 188–189
rowversion, 190
spatial types, 190
sql_variant, 190
uniqueidentifier, 190
user defined type/alias, 193–194
UserHandle, 185–186, 195
varchar(N), 195
varchar(max) column, 195
XML, 190
documentation, 185
ease of implementation, 184
logical modeling, 184
nullability selection, 198
SurrogateKey, 185
TopicName, 185
userHandle, 200
Domains
consistent model, 64
definition, 63
GUID value, 64
logical modeling, 64, 65
modeling, automated tool, 63
standard, 65
subclasses, 63
type, 64
types of person, 65
driversLicenseNumber (driversLicense), 151
driversLicenseStateCode, 151
driversLicenseStateCode (driversLicense), 151

E

EAV. *See* Entity-attribute-value
Education, 1
EmailAddress, 132–133, 156
EmailAddressNumber, 156
employeeNumber, 151, 178
Encapsulation, 614–616
EndDate, 175
Entity-attribute-value (EAV)
dynamic statement, 351
EquipmentPropertyType table creation, 348–349
EquipmentType column, 352
property schema, 348
property table, 347
row insertion, 349
sql_variant column format, 351–352
trappable error, 349–351
Entity-relationship (E-R) modeling method, 41
errorHandlingTestId, 293
Error logging procedure, 590–593
ExampleModel, 146
EXCEPT relational operator, 570
Extended DDL utilities, 588–590
EyeColor, 146

F

Filestream storage, 193, 332
FirstName, 151, 171
First Normal Form
column names with numbers at end, 143
string data with separator-type characters, 143
tables with no/poorly-defined keys, 144
float(N) datatype, 189
fromMessgingUser, 229

G

GameInstance, 273
GamePlatform, 273
Game tables, 274
GloballyUniqueIdentifier (GUID), 212
key, 187
value, 64
GrandparentName, 151

H

heirarchyId, 195
Hierarchies
hierarchyTypeId type, 326–329
query optimizations
Kimball helper table, 330–332
nested sets, 330–331
path technique, 330
self-referencing relationship
graphs, 325–326
trees, 320–325 (*see also* Single-parent hierarchies)
Highly recognized abbreviations, 175

I

Identifiers
candidate keys, 111
Downtown Office, 112

www.it-ebooks.info

Identifiers (*cont.*)
element, 111
Office entity, 112
Patient entity, 113
people's name, 112
IDENTITY property, 209–210
Indexes, 214, 217–218, 459
clustered indexes, 462–463, 478
ALTER INDEX REORGANIZE, 473
clustering key, 472
equality and inequality, 474
GUID, 472
NEWSEQUENTIALID() function, 472
OLTP setting, 473
range queries, 472
scan, 473
SET STATISTICS IO, 474
creation, 468–470
dynamic management view queries
fragmentation, 501
index utilization statistics, 500–501
missing indexes, 497–499
foreign keys
domain tables, 490, 491
indexed views, 493–496
many-to-many resolution table relationships, 492
OLTP database, 489
one-to-one relationships, 493
ownership relationships, 491–492
sample relationship, 489
siteClickLog table, 489, 490
lookup, 471
Management Studio, 471
nonclustered indexes
abstract representation, 464
AVG_RANGE_ROWS, 47
clustered tables, 465–466
columnstore index, 464
composite indexes, 481–484
covering indexes, 484–485
CREATE TABLE testIndex, 479
DBCC SHOW_STATISTICS command, 477, 478
DELETE operation, 476
DISTINCT_RANGE_ROWS, 479
dynamic management views, 475–477
EQ_ROWS, 479
filtered index, 481
heap, 466–467, 487–488
histogram, 479, 481
index keys, sort order, 486–487
INSERT operation, 476
leaf page, 464
multiple indexes, 485–486
pointer, 464
ProductId column, 478
profiler, 475
queries density, 478
query processor, 480
RANGE_HI_KEY, 479
RANGE_ROWS, 479
row locator, 464
sample index, 464
SELECT operation, 476
slow queries, 475
TABLESAMPLE clause, 480
UPDATE operation, 476
User lookups, 477
User scans, 477
User seeks, 477
User updates, 477
rules, 501–503
scan, 471
seek, 471
SET SHOWPLAN_TEXT ON statement, 470
structure, 460–462
unique indexes, 488
INSTEAD OF triggers
applications, 728
perform action comment, 727–728 (insert symbol in word file)
DBMS, 284
error handler, 730
formatting user input, 284–287
FormatUpper value, 730
insert statement, 729–730
multi-row operations, 730
no action on table, 290–292
redirecting invalid data, 287–290
RowCreatedTime and RowLastModify time values, 730–731
SCOPE_IDENTITY() function, 732–733
UPDATE statement, 731
int datatype, 188
Integration Definition for Information Modeling (IDEFIX), 54
InvoiceAmount, 152, 164
Isolation levels
application locks, 535
data consistency, 533
definition, 533
hypothetical code snippet, 533
nonrepeatable reads, 534–535
phantom rows, 534–535
READ COMMITTED, 535, 537–539
READ COMMITTED SNAPSHOT, 545–546
READ UNCOMMITTED, 536–537
REPEATABLE READ, 534, 539–540
REPEATABLE READ isolation level, 535

www.it-ebooks.info

SERIALIZABLE, 540–541
SNAPSHOT, 542–545
SNAPSHOT isolation level, 535
syntax, 535
sys.dm_exec_sessions, 536
IsSpecialSale, 191

K

Key implementation
alternate keys
constraints, 183
logical model, 182
Message table, 183
MessageTopic table, 183
messaging model, 183, 184
MessagingUser table, 182, 183
PartID, 182
PartNumber, 182
TopicId, 184
UserDefinedTopicName column, 184
primary key
existing columns, 178
new surrogate value, 178–182
Kimball helper table, 330–332

L

Large-value datatype columns, 192–193
LastName, 151, 171
Logs.Call, 282

M

Manufacturer table, 178
Messages.Message table, 221
MessageTime, 172, 197, 221
MessageTopic table, 188
MessagingUser, 177, 192
MessagingUserId columns, 198
MessagingUser table, 192
MiddleName, 151
Migrated key, 55
MovieRentalPackage table, 338, 339
Multiparent hierarchies, 325–326
Multiple active result sets (MARS), 507

N

nchar, nvarchar, nvarchar(max) datatype, 189
Nested sets, 375
Nonnatural primary keys
DDL, tables building, 212–214
default constraint, 210–212
IDENTITY property, 209–210
manual management, 209
Normalization, 129
BCNF, 144–146
dependency between rows, 152
entire key dependency, 147–149
forms, 130
multiple columns with same prefix, 153
partial key dependency, 146–147
positional meaning, 155–156
process, 130–131
relational database system, 129
repeating groups of data, 154
Second Normal Form, 144
SQL, 129
summary data, 154–155
surrogate keys effect, dependency
DoorCount, 149
driversLicense and employee tables with improper normalization, 150
driversLicense and employee tables with natural keys, 149
driversLicense and employee tables with surrogate keys, 150
driversLicense table, 151
GrandparentId, 151
Grandparent table, 151
key migration, 151
Parent table, 151
StateCode, 150
VehicleType, 149
VehicleTypeId, 149
table and column shape (*see* Tables)
tables with multiple meanings
Fifth Normal Form, 159–162
Fourth Normal Form, 157–159
Third Normal Form, 144
T-SQL, 129
Nullability, 198
Numbers table
additional attributes, 565
comma delimited items separation, 568, 571–573
D5 and D6 tables, code break, 565–566
definition, 564
DivisibleByNineFlag, 567
DivisibleBySevenFlag, 567
EXECUTE and SELECT clause, 567
integer cubed calculation, 573–574
non-negative integers, 564–565
numbers boundary, 575
precalculated and stored number sequence, 565
SELECT clause multiplier, 566
sequence gaps determination, 568, 570
string content determination, 568–570

www.it-ebooks.info

Numbers table (*cont.*)
sum of cubes, 574–575
taxicab numbers, 573
tempdb space, 575
WHERE condition, 574

O

Object relational mapping (ORM) tools, 181
OptInFlag, 175

P, Q

Parallelism, 507
ParentName, 151
Pascal-cased names, 176
Patterns, 301
data-driven design, 319–320
hierarchies (*see* Hierarchies)
images, documents, and other files
AccountFileDirectory, 340
API, 338
binary data, 332
binary format data storage, 339
binary storage, SQL Server's storage engine, 332
CREATE TABLE TestSimpleFileStream, 334
customer table with picture, 339
directory creation, 335
DIRECTORY_NAME parameter, 335
encryption, 338
files location, 338
filestream access, 333
filestream attribute, 335
filestream column, 333, 334
filestream data, 334
filestream files, 334
filetable access, 335
FILETABLE_DIRECTORY, 335
filetable directory in Windows Explorer, 337
FileTableRootPath() function, 336
filetable style, 335
generalization, 340–345
image and data backup, 338
MovieRentalPackage table with PictureUrl datatype, 339
MSSQLSERVER, 335
NON_TRANSACTED_ACCESS, 335
path reference, file data, 332
Remote Blob Store API, 337
ROWGUIDCOL property, 334
security, 338
size, 338
SQL tables, 337
stream_id, 336, 340
transactional integrity, 333, 338, 340
unstructured file data, 340
User-specified data storage (*see* User-specified data storage)
utilization, 384
varbinary(max) column, 332, 333
uniqueness (*See* Uniqueness, patterns)
Patternsdata-driven design, 302
Patternsgeneralization, 302
Patternshierarchies, 302
Patternsimages, documents, and other files, 302
Patternsstoring user-specified data, 302
Patternsuniqueness, 302
PersonId, 175
PersonName, 175
PersonNumber, 175
PersonSocialSecurityNumber, 175
Physical database structure
allocation map, 451
bulk changed map, 451
CREATE INDEX command, 456
data pages, 451
data row, 453
datatype-level compression, 455
dictionary compression, 455
differential changed map, 451
extents, 450–451
files and filegroups
ALTER TABLE statement, 448
COLLATE database_default, 450
CONCAT function, 450
database storage organization, 447
default filegroup, 448
DROP_EXISTING setting, 448
FILEGROWTH parameter, 449
hyperthreading, 448
I/O load distribution, 447
primary files, 448
secondary files, 448
sys.filegroups catalog view, 449–450
index allocation and data, 451
logical level, 3
mixed extent, 451
overflow data and page, 451, 452
page compression, 455
page free space, 451
page splits, 454–455
partitioning, 456–459
physical level, 4
prefix compression, 455
row compression, 455
table data, 451
uniform extent, 451
Physical Data Independence rule, 446

www.it-ebooks.info

Physical model implementation, 169
adding implementation columns, 201–202
database generation tools, 242
database requirements, 170–171
datatype and nullability, 242
DDL (*See* Data definition language (DDL))
Developer Edition SQL Server, 170
document and script, 242
domain implementation (*See* Domain implementation)
Enterprise Evaluation Edition SQL Server, 170
foreign key constraints, 242
key implementation (*See* Key implementation)
logical database design, 171
name selection
delimited identifiers, 173
fred]olicious, 173
identifiers, 172
model name adjustments, 176–177
naming columns, 175–176
naming tables, 173–174
policy-based management, 173
regular identifiers, 172–173
sysname, 172
normalization maintanance, 242
procedure, 169–170
real strategy for naming objects, 242
schemas, 200–201
SQL Server Express, 170
structure testing
CATCH block, 240
delete statements, 238
foreign key constraints, 237
integration testing, 237
MessageTime, 240
MessagingUser table, 238–239
non-alphanumeric character, 239
RAISERROR statement, 238
RoundedMessageTime, 240
semi-significant issues, 241
six-table database, 241
Sourceforge, 237
test scripts, 237
user defined topic, 240–241
user handle, 239
tables and columns documentation
Message, 172
MessageTopic, 172
Topic, 172
User, 171
UserConnection, 172
template domains, 242
test script, 242
UNIQUE constraint, 242
PledgeAmount, 175
Policy-based management, 173
Precise numeric data
decimal values, 676
float and real datatypes, 676
money types, 678–680
SET NUMERIC_ROUNDABORT ON, 677
testvar decimal, 677
integer values, 673–674
bigint, 676
int, 675
smallint, 674–675
tinyint, 674
PreferredCustomerFlag, 192
PreferredCustomerIndicator, 192
Primary key
existing columns, 178
new surrogate value
advantages, 179
disadvantages, 179
human-accessible key, 181
IDENTITY property, 179
messaging Database Model, 181
MessagingUser table, 180–181
ORM tools, 181
otherColumnsForAltKey, 179
parentKeyValue, 179
single-column key, 178, 181
UNIQUE constraint, 179
Primary key (PK), 19
Primary limiting factor, 575
productNumber, 178
ProductPrice column, 152
Profiler, 441–442
Pronounced abbreviations, 175
PublisherLocation, 145, 153
PublisherName, 153
PurchaseOrderNumber, 175

R

Ralph Kimball's approach, 640
Range uniqueness
appointment, 317–318
appointmentId, 314
bad row removal, 315
data testing qurey, 314–315
delete operation, 315–316
doctorId number, 316
improper data situation, 313
table creation, 313–314
real datatype, 189
Recursive algorithm, 320
Regular identifiers, 172–173

www.it-ebooks.info

Relational database design, 1
binary relationship, 26
cardinality, 25–26
Codd's rule, RDBMS, 3
comprehensive data sublanguage, 5–6
distribution independence, 8
dynamic online catalog, 5
guaranteed access, 4
high-level insert, update and delete, 6
information principle, 3–4
integrity independence, 7–8
logical data independence, 7
nonsubversion rule, 8
NULL values, 4–5
physical data independence, 6–7
view table, 6
database-specific project phase, 32
conceptual, 33–34
fundamental techniques, 38
logical, 34
physical, 35
storage, 35
data structure, 2
dependency
determinants, 31
functional, 30–31
entity relationship, 2, 25
foreign key, 24
history, 2–3
many-to-many relationship, 29
nonbinary relationship, 29–30
one-to-exactly N relationship, 28–29
one-to-many relationship, 27–28
parent and child tables, 25
relational programming, 2, 31–32
SQL standards, 8–9
structure recognition, 9
databases and schemas, 10
domain definition, 16–17
key (*see* Candidate key)
metadata storage, 17–18
missing (NULL) values, 15–16
tables, rows and columns (*see* Tables)
Reporting design, 639
aggregation querying, 667
indexing, 669–670
queries, 668–669
analytical querying
indexing, 663–664
queries, 661–663
requirements-gathering process, 641–642
styles
aggregation reporting (*see* Aggregation reporting style)
analytical reporting. (*see* Analytical reporting style)
innies and outties, definition, 639
Request for proposal (RFP), 49
Request for quote (RFQ), 49
Resource Governor, 500
Reusable standard database components, 563
calendar table (*see* Calendar table)
error logging procedure, 564, 590–592, 593
numbers table, 564, 593
additional attributes, 565
comma delimited items separation, 568, 571–573
D5 and D6 tables, code break, 565–566
definition, 564
DivisibleByNineFlag, 567
DivisibleBySevenFlag, 567
EXECUTE and SELECT clause, 567
integer cubed calculation, 573–574
non-negative integers, 564–565
numbers boundary, 575
precalculated and stored number sequence, 565
SELECT clause multiplier, 566
sequence gaps determination, 568, 570
string content determination, 568–570
sum of cubes, 574–575
taxicab numbers, 573
tempdb space, 575
WHERE condition, 574
OLTP style load, 563
reference/demographic information, 592
security, 592
universal data models, 563
user-defined functions, 592
utility objects, 564, 593
extended DDL utilities, 586, 588–590
monitoring tools, 586–588
SQL Server, 585
temp usage, 586
Reverse cascade operation, 275
RoundedMessageTime, 172, 183, 197
RowCreateTime, 201, 218, 226–228
RowLastUpdateTime, 201, 218, 226–228
ROW_NUMBER() windowing function, 140
rowversion datatype, 190, 202

S

Sales table, 155
SaveTime, 175
Scalar datatype
approximate numeric data, 680–681
binary data, 692–693

www.it-ebooks.info

binary length, 693–694
image, 695
varbinary length, 694
varbinary max, 694
bit, 695–696
character strings, 688
char length, 688–689
nchar, nvarchar, ntext, 691–692
text, 691
varchar length, 689–690
varchar max, 690–691
cursor, 700
date, 681–682
date functions, 685–686
date ranges, 686
date representation, text formats, 687–688
datetime, 684
datetime2, 682–683
datetimeoffset, 683
non-scalar types, 704–705
precise numeric data (*see* Precise numeric data)
rowversion, 696–697
smalldatetime, 683–684
sql_variant, 703–704
table valued parameters, 702–703
table variables, 700–702
time, 682
uniqueidentifier, 697–699
SCOPE_IDENTITY(), 226
Selective uniqueness
customerSettings table, 303
duplicate values, 306
error messages, 306
EXISTS query, 305
filtered index, 303, 304, 342
human resources employee table, 303
indexed views, 303, 306
initial sample row creation, 304
insurancePolicyNumber, 304, 306
NULL, 304
policy numbers, 303
primaryContactFlag, 305
row with 'fred,' 305
two rows with null, 304
SELECT SCOPE_IDENTITY(), 287
SentToUserHandle, 172
Server and database audit
audit configuration, 433–434
audit trail, 432–433
queue mechanisms, 430
service broker queues, 430
specification, 430–432
SQL server audit, 430
structure, 430
ShippedToEmailAddress, 156
Single-parent hierarchies
adjacency list, 322
children position, 324
circular references, 325
demonstration company hierarchy, 321
functional language, 320
parents of rows, 324
procedural language, 320
query code, children, 323–324
recursive type query, 325
single-instance-at-a-time access, 321
table creation, 322–323
tree structure searched depth first, 321
tree structure with levels, 321
Slowly changing dimension
isCurrent flag, 649–650
Member dimension table, 648–649
preferred query, 650
Provider dimension table, 650–651
rejected query, 650–652
start date and end date, 650
types, 647
smalldatetime, 189
smallint datatype, 188
Smart keys, 22
Snapshot fact, 659–661
Snowflake dimension, 652û653
Sorta Preferred Customer, 192
Spatial datatype, 190
SpecialSaleFlag, 191
SQL server audit, 430
SQL Server code, 518
stored procedures
CATCH block, 521
doomed transaction, 521
error, 519
error handling, 520–521
object_name function, 520
RAISERROR, 521, 524
ROLLBACK TRANSACTION statement, 519
savepoint, 520, 522
schema, 522–524
THROW statement, 524
triggers, 524–527
SQL Server Management Studio, 597
SQL-86 standard, 8
SQL_variant datatype, 190
StoreId, 175
Structure English Query Language (SEQUEL), 8
SUBSTRING function, 192, 571–572
SurrogateKey domain, 141, 196
SYNDATETIME() function, 227, 228
sysname, 172

www.it-ebooks.info

T

Tables
- atmonic values, 11
- atomic columns
  - column names with numbers at end, 143
  - complex datatypes, 132
  - data-protection schemes, 132
  - different rows, 141–143
  - e-mail addresses, 132–135
  - names, 135–136
  - same rows values, 139–141
  - string data with separator-type characters, 143
  - tables with no/poorly-defined keys, 144
  - telephone numbers, 136–138
- basic data representation terms, 13
- column term breakdown, 14
- Excel table, 11
- First Normal Form, 131
- information storage, 11
- record manager, 12
- relational theory, 12
- requirements, 131
- row term breakdown, 14
- scalar value, 11

Text, 172
TickerCode, 175
time datatype, 189
timestamp datatype, 190
tinyint datatype, 188
Tools.DivisibleByNineAndSevenNumber table, 567
TopicName, 172
Topic table, 188
Transaction fact
- bank transaction, 655
- Claim Payment fact table, 656
- degenerate dimensions, 659
- foreign keys, 656–657
- preferred query, 657
- rejected query, 657–658

Transact-SQL (T-SQL), 225, 226, 596
- brute force algorithm, 632
- CLR (*see* CLR)
- comma-delimited value, 633
- DBAs, 629
- flow language, 628, 631
- GREATEST function, 631–632
- guidelines, 634–635
- .NET language, 629
- SELECT clause, 631
- tempdb object, 632
- time compiler, 632

Transparent data encryption (TDE), 373
T-SQL. *See* Transact-SQL
Type dimension, 653–655
TypeOfAttendee, 171, 177

U

Unified Modeling Language (UML), 41
uniqueidentifier datatype, 190
Uniqueness, patterns
- approximate, 318–319
- bulk
  - baseplate types, 311
  - bricks types, 311
  - BuildInstance, 308, 311
  - BuildInstancePiece, 311
  - CTE, 312
  - data loading, 309
  - grouping sets, 311–312
  - InventoryAdjustment Quantity, 307
  - InventoryAdjustment table, 306
  - inventory and utilization storage, 306
  - inventory model, 307
  - Lego® collection, 307
  - minimal builds, 310
  - personal inventory, 308
  - Piece, 311
  - Piece table, 312–313
  - ProductSale Quantity, 307
  - row constructor syntax, 310
  - rows and data manipulation, 306
  - sample Lego parts, 307
  - table creation, 308
  - table implementation, 309
  - types of pieces, 309–310
- range
  - appointment, 317–318
  - appointmentId, 314
  - bad row removal, 315
  - data testing qurey, 314–315
  - delete operation, 315–316
  - doctorId number, 316
  - improper data situation, 313
  - table creation, 313–314
- selective
  - customerSettings table, 303
  - duplicate values, 306
  - error messages, 306
  - EXISTS query, 305
  - filtered index, 303, 304
  - human resources employee table, 303
  - indexed views, 303, 306
  - initial sample row creation, 304
  - insurancePolicyNumber, 304, 306
  - NULL, 304
  - policy numbers, 303

www.it-ebooks.info

primaryContactFlag, 305
row with 'fred,' 305
two rows with null, 304
types, 302–303
User-defined functions (UDF), 248, 255, 257
UserDefinedTopicName, 172
User-defined types (UDTs), 193–194
UserHandle, 171, 172, 193, 194
UserId, 183
User interface (UI), 37
UserName, 175
User-specified data storage
adding columns to table
ALTER TABLE, 353
CHECK constraint, 355
column set, 354, 357
HammerHeadStyle, 356
invalid value error, 356
Length column, 354–355
non-administrative users, 353
normal UPDATE statement, 355
primary key, 353
SELECT statement, 356
sparse columns, 353–354, 357–359
big old list, generic columns, 347
EAV
dynamic statement, 351
EquipmentPropertyType table creation, 348–349
EquipmentType column, 352
property schema, 348
property table, 347
row insertion, 349
sql_variant column format, 351–352
trappable error, 349–351
Equipment table, 346
methods, 346
T-SQL flexibility, 345

V

varbinary(max) datatype, 189, 193, 194
varbinary(N) datatype, 189
varchar(max) datatype, 189, 192, 193
varchar(N) datatype, 189
VehicleBrand, 159, 161
VehicleStyle, 146, 157, 159, 161

W

Waterfall method, 38
Website-linking system, 270, 271

X, Y, Z

XML
datatype, 190
hierarchyId, 673

www.it-ebooks.info

Pro SQL Server 2012 Relational Database Design and Implementation

Louis Davidson
with Jessica M. Moss

Apress®

www.it-ebooks.info

Pro SQL Server 2012 Relational Database Design and Implementation

Copyright © 2012 by Louis Davidson with Jessica M. Moss

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-3695-5

ISBN-13 (electronic): 978-1-4302-3696-2

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

President and Publisher: Paul Manning
Lead Editor: Richard Carey
Technical Reviewer: Rodney Landrum
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editors: Jessica Belanger and Stephen Moles
Copy Editor: Heather Lang
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

www.it-ebooks.info

This book is dedicated my mom.

–Louis

www.it-ebooks.info

Contents

Foreword xix

About the Author xxi

About the Technical Reviewer xxiii

Acknowledgments xxv

Introduction xxvii

Chapter 1: The Fundamentals 1

Taking a Brief Jaunt Through History 2

Introducing Codd's Rules for an RDBMS 3

Nodding at SQL Standards 8

Recognizing Relational Data Structures 9

Introducing Databases and Schemas 10

Understanding Tables, Rows, and Columns 10

Working with Missing Values (NULLs) 15

Defining Domains 16

Storing Metadata 17

Assigning Uniqueness Constraints (Keys) 18

Understanding Relationships 24

Working with Binary Relationships 26

Working with Nonbinary Relationships 29

Understanding Dependencies 30

Working with Functional Dependencies 30

Working with Determinants 31

www.it-ebooks.info

Relational Programming 31
Outlining the Database-Specific Project Phases 32
Conceptual Phase 33
Logical Phase 34
Physical 35
Storage Phase 35
Summary 35
Chapter 2: Introduction to Requirements 37
Documenting Requirements 39
Gathering Requirements 41
Interviewing Clients 42
Asking the Right Questions 43
What Data Is Needed? 43
How Will the Data Be Used? 44
What Rules Govern the Use of the Data? 44
What Data Is Reported On? 45
Where Is the Data Now? 46
Will the Data Need to Be Integrated with Other Systems? 47
How Much Is This Data Worth? 47
Who Will Use the Data? 47
Working with Existing Systems and Prototypes 48
Utilizing Other Types of Documentation 48
Early Project Documentation 49
Contracts or Client Work Orders 49
Level of Service Agreement 49
Audit Plans 50
Following Best Practices 50
Summary 50

www.it-ebooks.info

Chapter 3: The Language of Data Modeling 53
Introducing Data Modeling 54
Entities 55
Attributes 58
Primary Keys 59
Alternate Keys 61
Foreign Keys 62
Domains 63
Naming 65
Relationships 67
Identifying Relationships 68
Nonidentifying Relationships 69
Role Names 71
Relationship Cardinality 72
Verb Phrases (Relationship Names) 79
Descriptive Information 81
Alternative Modeling Methodologies 82
Information Engineering 83
Chen ERD 85
Visio 86
Management Studio Database Diagrams 87
Best Practices 88
Summary 89
Chapter 4: Initial Data Model Production 91
Example Scenario 92
Identifying Entities 93
People 94
Places 95
Objects 95

www.it-ebooks.info

Ideas 96
Documents 97
Groups 98
Other Entities 99
Entity Recap 100
Relationships between Entities 102
One-to-N Relationships 102
Many-to-Many Relationships 106
Listing Relationships 107
Identifying Attributes and Domains 109
Identifiers 111
Descriptive Information 113
Locators 113
Values 115
Relationship Attributes 116
A List of Entities, Attributes, and Domains 117
Identifying Business Rules 120
Identifying Fundamental Processes 122
The Intermediate Version of the Logical Model 124
Identifying Obvious Additional Data Needs 124
Review with the Client 125
Repeat Until the Customer Agrees with Your Model 126
Best Practices 126
Summary 127
Chapter 5: Normalization 129
The Process of Normalization 130
Table and Column Shape 131
All Columns Must Be Atomic 131
All Rows Must Contain the Same Number of Values 139

www.it-ebooks.info

All Rows Must Be Different 141
Clues That an Existing Design Is Not in First Normal Form 143
Relationships Between Columns 144
BCNF Defined 144
Partial Key Dependency 146
Entire Key Dependency 147
Surrogate Keys Effect on Dependency 149
Dependency Between Rows 151
Clues That Your Database Is Not in BCNF 152
Positional Meaning 155
Tables with Multiple Meanings 156
Fourth Normal Form: Independent Multivalued Dependencies 157
Fifth Normal Form 159
Denormalization 162
Best Practices 165
Summary 165
The Story of the Book So Far 167
Chapter 6: Physical Model Implementation Case Study 169
Choosing Names 172
Table Naming 173
Naming Columns 175
Model Name Adjustments 176
Choosing Key Implementation 177
Primary Key 177
Alternate Keys 182
Determining Domain Implementation 184
Implement as a Column or Table? 186
Choosing the Datatype 188

www.it-ebooks.info

Choosing Nullability 198
Choosing a Collation 199
Setting Up Schemas 200
Adding Implementation Columns 201
Using DDL to Create the Database 202
Creating the Basic Table Structures 204
Adding Uniqueness Constraints 214
Building Default Constraints 218
Adding Relationships (Foreign Keys) 219
Adding Basic Check Constraints 224
Triggers to Maintain Automatic Values 226
Documenting Your Database 230
Viewing the Basic Metadata 233
Unit Testing Your Structures 237
Best Practices 241
Summary 242
Chapter 7: Data Protection with Check Constraints and Triggers 245
Check Constraints 247
CHECK Constraints Based on Simple Expressions 252
CHECK Constraints Using Functions 254
Enhancing Errors Caused by Constraints 258
DML Triggers 260
AFTER Triggers 261
Relationships That Span Databases and Servers 279
INSTEAD OF Triggers 283
Dealing with Triggers and Constraints Errors 292
Best Practices 297
Summary 298

www.it-ebooks.info

Chapter 8: Patterns and Anti-Patterns 301

Desirable Patterns 302

Uniqueness 302

Data-Driven Design 319

Hierarchies 320

Images, Documents, and Other Files, Oh My 332

Generalization 340

Storing User-Specified Data 345

Anti-Patterns 359

Undecipherable Data 360

One-Size-Fits-All Key Domain 361

Generic Key References 364

Overusing Unstructured Data 367

Summary 368

Chapter 9: Database Security and Security Patterns 371

Database Access Prerequisites 372

Guidelines for Server Security 373

Principals and Securables 374

Connecting to the Server 375

Using Login and User 376

Using the Contained Database Model 379

Impersonation 383

Database Securables 386

Grantable Permissions 387

Controlling Access to Objects 388

Roles 392

Schemas 400

www.it-ebooks.info

Controlling Access to Data via T-SQL–Coded Objects 402
Stored Procedures and Scalar Functions 402
Impersonation within Objects 404
Views and Table-Valued Functions 410
Crossing Database Lines 417
Using Cross-Database Chaining 418
Using Impersonation to Cross Database Lines 423
Using a Certificate-Based Trust 424
Different Server (Distributed Queries) 426
Obfuscating Data 427
Monitoring and Auditing 429
Server and Database Audit 430
Watching Table History Using DML Triggers 434
DDL Triggers 438
Logging with Profiler 441
Best Practices 442
Summary 443
Chapter 10: Table Structures and Indexing 445
Physical Database Structure 447
Files and Filegroups 447
Extents and Pages 450
Data on Pages 453
Partitioning 456
Indexes Overview 459
Basic Index Structure 460
Index Types 462
Clustered Indexes 462
Nonclustered Indexes 463

www.it-ebooks.info

Nonclustered Indexes on Clustered Tables 461
Nonclustered Indexes on a Heap 462
Basics of Index Creation 464
Basic Index Usage Patterns 466
Using Clustered Indexes 467
Using Nonclustered Indexes 471
Using Unique Indexes 484
Advanced Index Usage Scenarios 484
Indexing Foreign Keys 489
Indexing Views 493
Index Dynamic Management View Queries 497
Missing Indexes 497
Index Utilization Statistics 500
Fragmentation 501
Best Practices 501
Summary 503
Chapter 11: Coding for Concurrency 505
What Is Concurrency? 506
OS and Hardware Concerns 508
Transactions 509
Transaction Syntax 510
Compiled SQL Server Code 518
Isolating Sessions 527
Locks 528
Isolation Levels 533
Coding for Integrity and Concurrency 546
Pessimistic Locking 546
Implementing a Single-Threaded Code Block 549
Optimistic Locking 552

www.it-ebooks.info

Row-Based Locking 553
Logical Unit of Work. 558
Best Practices 560
Summary 561
Chapter 12: Reusable Standard Database Components ..... 563 Numbers Table 564
Determining the Contents of a String 568
Finding Gaps in a Sequence of Numbers 570
Separating Comma Delimited Items 571
Stupid Mathematic Tricks 573
Calendar Table 576
Utility Objects 585
Monitoring Objects 586
Extended DDL Utilities 588
Logging Objects 590
Other Possibilities... 592
Summary 593
Chapter 13: Considering Data Access Strategies ..... 595 Ad Hoc SQL 597
Advantages 597
Pitfalls. 607
Stored Procedures. 613
Encapsulation 614
Dynamic Procedures. 616
Security 619
Performance 621
Pitfalls. 623
All Things Considered...What Do I Choose? 627

www.it-ebooks.info

T-SQL and the CLR 629
Guidelines for Choosing T-SQL 633
Guidelines for Choosing a CLR Object 634
CLR Object Types 634
Best Practices 637
Summary 638
Chapter 14: Reporting Design 639
Reporting Styles 639
Analytical Reporting 640
Aggregation Reporting 641
Requirements-Gathering Process 641
Dimensional Modeling for Analytical Reporting 642
Dimensions 644
Facts 655
Analytical Querying 661
Queries 661
Indexing 663
Summary Modeling for Aggregation Reporting 664
Initial Summary Table 665
Additional Summary Tables 667
Aggregation Querying 667
Queries 668
Indexing 669
Summary 670
Appendix A 671
Appendix B 707

Index 735

www.it-ebooks.info

Foreword

When Louis asked me to write the foreword to this book, I thought he was joking. Why would anyone want a developer to write the foreword for a database book? A quick reminder from Louis made it all clear—I mention the predecessor to this book in my consulting engagements. Who better to demonstrate how effectively he communicates the concepts than someone who lives and breathes databases everyday? Well, I am here to tell you that if you are looking for a sound technical resource for working with SQL Server, look no further.

Once again, Louis has done a remarkable job turning the critical details of SQL Server into an easy-to-read style that is more of a conversation than a technical manual. Louis relates what you really need to know based on his considerable experience and insight. I'm proud to have the opportunity to get to know Louis through his community endeavors and to leverage his knowledge to save time in delivering some of my solutions.

—John Kellar
Chairman, Devlink Technical Conference, and Microsoft MVP

www.it-ebooks.info

About the Author



Louis has been in the IT industry (for what is starting to seem like a really long time) as a corporate database developer and architect. He has been a Microsoft MVP for eight years and this is the fifth edition of this database design book. Louis has been active speaking about database design and implementation at many conferences over the past ten years, including SQL PASS, SQL Rally, SQL Saturday events, CA World, and the Devlink developer conference. Louis has worked for the Christian Broadcasting Network as a developer, DBA, and data architect, supporting offices in Virginia Beach, Virginia, and in Nashville, Tennessee, for over 14 years. Louis has a bachelor's degree from the University of Tennessee at Chattanooga in computer science.

For more information please visit his web site at drsql.org.



Jessica M. Moss is a well-known practitioner, author, and speaker of Microsoft SQL Server business intelligence. She has created numerous data warehouse and business intelligence solutions for companies in different industries and has delivered training courses on Integration Services, Reporting Services, and Analysis Services. While working for a major clothing retailer, Jessica participated in the SQL Server 2005 TAP program where she developed best implementation practices for Integration Services. Jessica has authored technical content for multiple magazines, websites, and books, and has spoken internationally at conferences such as the PASS Community Summit, SharePoint Connections, and the SQLTeach International Conference. As a strong proponent of developing user-to-user community relations, Jessica actively participates in local user groups and code camps in central Virginia. In addition, Jessica volunteers her time to help educate people through the PASS organization.

www.it-ebooks.info

About the Technical Reviewer



Rodney Landrum has been architecting solutions for SQL Server for over 12 years. He has worked with and written about many SQL Server technologies, including DTS, integration services, analysis services, and reporting services. He has authored three books on reporting services. He is a regular contributor to *SQL Server* Magazine, SQLServerCentral.com, and Simple-Talk.com. Rodney is also an SQL Server MVP.

www.it-ebooks.info

Acknowledgments

"I awoke this morning with devout thanksgiving for my friends, the old and the new."

—Ralph Waldo Emerson

I am not a genius, nor am I some form of pioneer in the database design world. I acknowledge that the following "people" have been extremely helpful in making this book happen along the way. Some help me directly, while others probably don't even know that this book exists. Either way, they have all been an important part of the process.

Far above anyone else, Jesus Christ, without whom I wouldn't have had the strength to complete the task of writing this book. I know I am not ever worthy of the love that you give me.

My wife, Valerie Davidson, for putting up with this craziness for a fifth time.

Gary Cornell, for giving me a chance to write the book that I wanted to write.

My current managers, Mark Carpenter, Andy Curley, and Keith Griffith, for giving me time to go to several conferences that really helped me to produce as good of a book as I did. All of my coworkers at CBN that provide me with many examples for this book and my other writing projects.

The PASS conferences (particularly SQL Saturday events), where I was able to hone my material and meet thousands of people over the past three years and find out what they wanted to know.

Jessica Moss, for teaching me a lot about data warehousing, and taking the time to write the last chapter of this book for you.

Paul Nielsen, for challenging me to progress and think harder about the relational model and its strengths and weaknesses.

The MVP Program, for giving me access to learn more about the internals of SQL Server over the years.

The fantastic editing staff I've had, including Jonathan Gennick who (figuratively) busted my lip a few times over my poor use of the English language and without whom the writing would sometimes appear to come from an illiterate chimpanzee. Most of these people are included on the copyright page, but I want to say a specific thanks to Tony Davis (who had a big hand in the 2005 version of the book) for making this book great, despite my frequently rambling writing style.

To the academics out there who have permeated my mind with database theory, such as E. F. Codd, Chris Date, Fabian Pascal, Joe Celko, my professors at the University of Tennessee at Chattanooga, and many others. I wouldn't know half as much without you. And thanks to Mr. Date for reviewing Chapter 1; you probably did more for the next version of this book than the current one.

All of the people I have acknowledged in previous editions that were so instrumental in getting this book where it is from all of the many changes and lessons over the years. I built upon the help you all provided over the past 12+ years.

Louis Davidson

www.it-ebooks.info