THE EXPERT'S VOICE® IN SQL

Pro SQL Server Relational Database Design and Implementation

Database design delivering business results

Fifth Edition

Louis Davidson
with Jessica Moss
Foreword by Kalen Delaney

Apress®

WOW! eBook
www.wowebook.org

Pro SQL Server Relational Database Design and Implementation

Fifth Edition

Louis Davidson
with Jessica Moss
Foreword by Kalen Delaney

Apress®

WOW! eBook
www.wowebook.org

Pro SQL Server Relational Database Design and Implementation

Louis Davidson
Antioch, Tennessee
USA

Jessica Moss
Mechanicsville, Virginia
USA

ISBN-13 (pbk): 978-1-4842-1972-0
ISBN-13 (electronic): 978-1-4842-1973-7
DOI 10.1007/978-1-4842-1973-7

Library of Congress Control Number: 2016963113

Copyright © 2016 by Louis Davidson

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

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.

Managing Director: Welmoed Spahr
Lead Editor: Jonathan Gennick
Development Editor: Laura Berendson
Technical Reviewers: Rodney Landrum, Arun Pande, Andy Yun, Alexzander Nepomnjashiy
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Todd Green, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Jill Balzano
Copy Editor: Bill McManus
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 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.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

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.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

WOW! eBook
www.wowebook.org

This book is dedicated to my mom, who passed away just after Thanksgiving in 2013. She had no idea what SQL was really, but was always encouraging and very proud of me as I was of her.

—Louis

WOW! eBook
www.wowebook.org

Contents at a Glance

Foreword xix

About the Authors xxi

About the Technical Reviewers xxiii

Acknowledgments xxv

Introduction xxvii

■ Chapter 1: The Fundamentals 1

■ Chapter 2: Introduction to Requirements 41

■ Chapter 3: The Language of Data Modeling 57

■ Chapter 4: Conceptual and Logical Data Model Production 97

■ Chapter 5: Normalization 141

■ Chapter 6: Physical Model Implementation Case Study 185

■ Chapter 7: Expanding Data Protection with Check Constraints and Triggers 271

■ Chapter 8: Patterns and Anti-Patterns 321

■ Chapter 9: Database Security and Security Patterns 411

■ Chapter 10: Index Structures and Application 491

■ Chapter 11: Matters of Concurrency 559

■ Chapter 12: Reusable Standard Database Components 627

■ Chapter 13: Architecting Your System 657

■ Chapter 14: Reporting Design 705

■ Appendix A: Scalar Datatype Reference 745

Index 779

WOW! eBook
www.wowebook.org

Contents

Foreword xix

About the Authors xxi

About the Technical Reviewers 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 17

Storing Metadata 18

Defining Uniqueness Constraints (Keys) 19

Understanding Relationships 25

Working with Binary Relationships 27

Working with Nonbinary Relationships 31

Understanding Functional Dependencies 32

Understanding Functional Dependencies 32

Finding Determinants 33

WOW! eBook
www.wowebook.org

Relational Programming 33
Outlining the Database-Specific Project Phases 34
Conceptual Phase 36
Logical Phase 36
Physical 37
Engine Adjustment Phase 37
Summary 38
Chapter 2: Introduction to Requirements 41
Documenting Requirements 43
Gathering Requirements 45
Interviewing Clients 46
Asking the Right Questions 47
Utilizing Other Types of Documentation 52
Early Project Documentation 53
Contracts or Client Work Orders 53
Level of Service Agreement 53
Audit Plans 53
Prototypes 54
Following Best Practices 54
Summary 54
Chapter 3: The Language of Data Modeling 57
Introducing Data Modeling 58
Entities 59
Attributes 62
Primary Keys 64
Alternate Keys 66
Foreign Keys 68
Domains 69

WOW! eBook
www.wowebook.org

Relationships 72
Identifying Relationships 73
Nonidentifying Relationships 74
Role Names 76
Relationship Cardinality 78
Verb Phrases (Relationship Names) 86
Descriptive Information 88
Alternative Modeling Methodologies 90
Information Engineering 90
Chen ERD 92
Best Practices 94
Summary 94
Chapter 4: Conceptual and Logical Data Model Production 97
Example Scenario 99
Building the Conceptual Model 99
Identifying Entities 100
Identifying Relationships Between Entities 109
Testing the Conceptual Model 119
Building the Logical Model 120
Identifying Attributes and Domains 120
Identifying Business Rules 132
Identifying Fundamental Processes 133
Finalizing the Logical Model 135
Best Practices 138
Summary 139
Chapter 5: Normalization 141
The Process of Normalization 142
Table and Column Shape 143
All Columns Must Be Atomic 143
All Rows Must Contain the Same Number of Values 152

WOW! eBook
www.wowebook.org

All Rows Must Be Different 155
Clues That an Existing Design Is Not in First Normal Form 157
Relationships Between Columns 158
BCNF Defined 158
Partial Key Dependency 160
Entire Key Dependency 162
Surrogate Keys Effect on Dependency 163
Dependency Between Rows 166
Clues That Your Database Is Not in BCNF 167
Positional Meaning 170
Tables with Multiple Meanings 171
Fourth Normal Form: Independent Multivalued Dependencies 172
Fifth Normal Form 175
Denormalization 178
Best Practices 180
Summary 181
The Story of the Book So Far 182
Chapter 6: Physical Model Implementation Case Study 185
Choosing a Physical Model for Your Tables 189
Choosing Names 191
Table Naming 192
Naming Columns 194
Model Name Adjustments 195
Choosing Key Implementation 197
Primary Key 197
Alternate Keys 202
Determining Domain Implementation 205
Enforce Domain in the Column, or With a Table? 207
Choosing the Datatype 209
Setting Nullability 219
Choosing the Collation 220

WOW! eBook
www.wowebook.org

Setting Up Schemas 222
Adding Implementation Columns 223
Using DDL to Create the Database 224
Creating the Basic Table Structures 227
Adding Uniqueness Constraints 238
Building Default Constraints 242
Adding Relationships (Foreign Keys) 243
Adding Basic Check Constraints 249
Triggers to Maintain Automatic Values 252
Documenting Your Database 255
Viewing the Basic System Metadata 259
Unit Testing Your Structures 263
Best Practices 268
Deployment Lifecycle 269
Summary 270
Chapter 7: Expanding Data Protection with Check Constraints and Triggers 271
Check Constraints 273
CHECK Constraints Based on Simple Expressions 279
CHECK Constraints Using Functions 281
Enhancing Errors Caused by Constraints 285
DML Triggers 287
AFTER Triggers 288
Relationships That Span Databases 303
INSTEAD OF Triggers 307
Dealing with Trigger and Constraint Errors 313
Best Practices 319
Summary 319

WOW! eBook
www.wowebook.org

Chapter 8: Patterns and Anti-Patterns 321

Desirable Patterns 322

Uniqueness 323

Data-Driven Design 340

Historical/Temporal Data 341

Hierarchies 356

Images, Documents, and Other Files, Oh My! 371

Generalization 380

Storing User-Specified Data 385

Anti-Patterns 398

Undecipherable Data 399

One-Size-Fits-All Key Domain 400

Generic Key References 403

Overusing Unstructured Data 406

Summary 408

Chapter 9: Database Security and Security Patterns 411

Database Access 413

Guidelines for Host Server Security Configuration 413

Principals and Securables 414

Connecting to the Server 416

Impersonation 425

Database Object Securables 428

Grantable Permissions 429

Roles 434

Schemas 442

Row-Level Security 444

Using Specific-Purpose Views to Provide Row-Level Security 446

Using the Row-Level Security Feature 448

Using Data-Driven Row-Level Security 454

WOW! eBook
www.wowebook.org

Controlling Access to Data via T-SQL–Coded Objects 455
Stored Procedures and Scalar Functions 456
Impersonation Within Objects 458
Views and Table-Valued Functions 465
Crossing Database Lines 467
Using Cross-Database Chaining 468
Using Impersonation to Cross Database Lines 474
Using a Certificate-Based Trust 475
Different Server (Distributed Queries) 477
Obfuscating Data 478
Encrypting Data 479
Using Dynamic Data Masking to Hide Data from Users 480
Auditing SQL Server Use 483
Defining an Audit Specification 484
Viewing the Audit Configuration 487
Best Practices 488
Summary 489
Chapter 10: Index Structures and Application 491
Indexing Overview 492
Basic Index Structure 493
On-Disk Indexes 496
Clustered Indexes 501
Nonclustered Indexes 507
Structure 507
Using the Nonclustered Index 511
Using Unique Indexes 526
Memory-Optimized Indexes 526
In-Memory OLTP Tables 527
Columnstore Indexes 536

WOW! eBook
www.wowebook.org

Common OLTP Patterns of Index Usage 537
When to Cluster on Something Other Than the PRIMARY KEY 537
Indexing Foreign Keys 538
Indexed Views 543
Compression 546
Partitioning 548
Indexing Dynamic Management View Queries 550
Missing Indexes 550
On-Disk Index Utilization Statistics 553
Fragmentation 554
In-Memory OLTP Index Stats 555
Best Practices 556
Summary 557
Chapter 11: Matters of Concurrency 559
OS and Hardware Concerns 562
Transactions 563
Transaction Overview 563
Transaction Syntax 565
SQL Server Concurrency Methods 577
Isolation Levels 577
Pessimistic Concurrency Enforcement 580
Optimistic Concurrency Enforcement 597
Coding for Asynchronous Contention 616
Row-Based Change Detection 617
Coding for Logical Unit of Work Change Detection 622
Best Practices 625
Summary 626

WOW! eBook
www.wowebook.org

Chapter 12: Reusable Standard Database Components 627

Numbers Table 628

Determining the Contents of a String 632

Finding Gaps in a Sequence of Numbers 634

Separating Comma-Delimited Items 635

Calendar Table 638

Utility Objects 647

Monitoring Objects 648

Extended DDL Utilities 650

Logging Objects 652

Other Possibilities… 654

Summary 655

Chapter 13: Architecting Your System 657

Choosing the Engine for Your Needs 659

Ad Hoc SQL 660

Advantages 661

Pitfalls 671

Stored Procedures 678

Advantages 679

Pitfalls 694

Stored Procedure or Ad Hoc? 698

T-SQL and the CLR 700

Best Practices 702

Summary 703

Chapter 14: Reporting Design 705

Reporting Styles 705

Analytical Reporting 706

Aggregation Reporting 706

Operational Reporting 707

WOW! eBook
www.wowebook.org

Requirements-Gathering Process 707
Dimensional Modeling for Analytical Reporting 708
Dimensions 710
Facts 721
Analytical Querying 727
Queries 727
Indexing 728
Summary Modeling for Aggregation Reporting 730
Initial Summary Table 731
Additional Summary Tables 733
Aggregation Querying 734
Queries 734
Indexing 735
Modeling (or Lack Thereof) for Operational Reporting 736
Sample Operational Data Model 737
In-Memory OLTP 737
Operational Querying 740
Queries 740
Indexing 740
Summary 743
Appendix A: Scalar Datatype Reference 745
Precise Numeric Data 747
Integer Values 747
Decimal Values 750
Approximate Numeric Data 754
Date and Time Data 755
date 755
time [(*precision*)] 755
datetime2 [(*precision*)] 756
datetimeoffset [(*precision*)] 756

WOW! eBook
www.wowebook.org

smalldatetime 757
datetime 757
Discussion on All Date Types 758
Character Strings 761
char[(*length*)] 761
varchar[(*length*)] 762
varchar(max) 763
text 764
Unicode Character Strings: nchar, nvarchar, nvarchar(max), ntext 764
Binary Data 765
binary[(*length*)] 765
varbinary[(*length*)] 766
varbinary(max) 766
image 767
Other Datatypes 767
bit 768
rowversion (aka timestamp) 768
uniqueidentifier 770
cursor 772
table 772
sql_variant 776
Not Simply Scalar Datatypes 778

Index 779

WOW! eBook
www.wowebook.org

Foreword

Database design is a hugely important topic, but one that is frequently overlooked. Because good database design is not dependent on a specific database product, vendors don't usually support education in design. As a long-time certified trainer, I was always well aware that issues pertaining to design were not covered in the standard courseware, because they are so vendor independent. For many years, and many versions, Microsoft had a course called "Implementing a Database Design", and people frequently shortened the name to "The DB Design Course", but it was about implementation, not design. It was about how to create the database after the design was done.

Proper design is not only important for data correctness, but it can also help you to troubleshoot effectively and isolate performance issues. Sometimes, little design changes can make a huge performance difference, but to understand what changes those might be, you need to understand not only design issues in general, but also the details of your system's actual design. And of course, the more you know, the better. But the ideal way to get the best performance from your database design is to start with a good design from the very beginning of a project.

So if no database vendor offers training courses in relational database design, how is one supposed to learn about this topic? Obviously, from books like the one you have here. Louis Davidson shows you not just how to implement a database design but how to first design your database from the stated requirements. In the first few chapters, Louis gives you a solid foundation in relational database concepts and data modeling, the difference between the logical and physical model, and the details of normalization. He goes on in the second part to show you how to take that design and create a database from it. Along the way he gives you the basics of several other database related concepts that can make an enormous difference in the performance (and thus the ultimate success) of your new database application: indexes and concurrency.

Modern relational database products have been built to allow you to create a database in minimal time and start loading data and writing queries very quickly. But if you want those queries to perform well and continue to give good performance as your database grows to gigabytes and perhaps even terabytes, you need a solid foundation of a good design. Time spent on database design is an investment with a huge return over the life of an application. You deserve to start out well, and Louis can help make that possible.

Kalen Delaney

www.SQLServerInternals.com

Poulsbo, WA – November 2016

WOW! eBook
www.wowebook.org

About the Authors

Louis Davidson has been working with databases (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 13 years and this is the fifth edition of this database design book with Apress. Louis has been active speaking about database design and implementation at many conferences over the past 14 years, including SQL PASS, SQL Rally, SQL Saturday events, CA World, and the devLink Technical Conference. Louis has worked for the Christian Broadcasting Network (CBN) as a developer, DBA, and data architect, supporting offices in Virginia Beach, Virginia, and in Nashville, Tennessee, for over 18 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 www.drsql.org.

Jessica Moss is a well-known practitioner, author, and speaker on 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, web sites, 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.

WOW! eBook
www.wowebook.org

About the Technical Reviewers

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.

Alexzander Nepomnjashiy is CIO at Kivach Clinic, www.kivach.ru.

WOW! eBook
www.wowebook.org

Andy Yun has been a database developer and administrator for SQL Server for almost 15 years. Leveraging knowledge of SQL Server Internals and extensive experience in highly transactional environments, he strives to make T-SQL leaner and meaner. Andy is extremely passionate about passing knowledge on to others, regularly speaking at User Groups, SQL Saturdays, and PASS Summit. Andy is a co-founder of the Chicago SQL Association, co-Chapter Leader of the Chicago Suburban SQL Server User Group, and part of the Chicago SQL Saturday Organizing Committee.

WOW! eBook
www.wowebook.org

Acknowledgments

You are never too old to set another goal or to dream a new dream.

—C. S. Lewis

I am not a genius, nor am I some form of pioneer in the database design world. I am just a person who 15 or so years ago asked the question of a publisher: "Do you have any books on database design?" The reply: "No, why don't you write one?" So I did, and I haven't stopped writing since then, with this book now in its fifth edition. I acknowledge that the following "people" have been extremely helpful in making this book happen and evolve along the way. Some have helped 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 yet another time, all while working on her doctorate in education.

Gary Cornell, for a long time ago 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. And to all of my coworkers at CBN, who have provided 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, and perhaps even more importantly, the MVPs and Microsoft folks I have come into contact with over the years. I have learned so much in the newsgroups, e-mail lists, and in particular, the MVP Summit that I could not have done half as well without them (check the first edition of this book for evidence that things got better after I became an MVP!).

WOW! eBook
www.wowebook.org

The fantastic editing staff I've had, including Jonathan Gennick and Jill Balzano, who (figuratively) busted my lip a few times over my poor structure, use of the English language, etc., 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 them and 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 in the previous edition; 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 who were so instrumental in getting this book to where it is from all of the many changes and lessons over the years. I've built upon the help you've all provided over the past 12+ years.

—Louis Davidson

WOW! eBook
www.wowebook.org

Introduction

It was a dark and stormy night…

—Snoopy

These words start many a work of fiction, and usually not the most believable works of fiction either. While this book is clearly, at its core, nonfiction, I felt the need to warn you that nearly every example in this book is fiction, carefully tailored to demonstrate some principle of database design. Why fictitious examples? My good friend, Jeremiah Peschka, once explained it perfectly when he tweeted: "@peschkaj: I'm going to demo code on a properly configured server with best practices code. Then you're all going to [complain] that it takes too long." The most egregious work of fiction will be the chapter on requirements, as in most cases the document to describe where to find the actual requirements documents will be longer than the chapter where I describe the process and include several examples.

So don't expect that if you can understand all of the examples in the book that you can easily be an expert in a week. The fact is, the real-world problems you will encounter will be far more complex, and you will need lots of practice to get things close to right. The thing you will get out of my book is knowledge of how to get there, and ideals to follow. If you are lucky, you will have a mentor or two who already know a few things about database design to assist you with your first designs. I know that when I was first getting started, I learned from a few great mentors, and even today I do my best to bounce ideas off of others before I create my first table. (Note that you don't need an expert to help you validate designs. A bad design, like spoiled milk, smells fonky, which is 3.53453 times worse than funky.)

However, what is definitely not fiction is my reason for writing (and rewriting) this book: that great design is still necessary. There is a principle among many programmers that as technology like CPU and disk improves, code needn't be written as well to get the job done fast enough. While there is a modicum of truth to that principle, consider just how wasteful this is. If it takes 100 milliseconds to do something poorly, but just 30 milliseconds to do it right, which is better? If you have to do the operation once, then either is just as good. But we don't generally write software to do something once. Each execution of that poorly written task is wasting 70 milliseconds of resources to get the job done. Now consider how many databases and how much code exist out there and guess what the impact to your slice of the world, and to the entire world, would be. "How good is good enough?" is a question one must ask, but if you aim for sleeping on the sidewalk, you are pretty much guaranteed not to end up with a mansion in Beverly Hills (swimming pools and movie stars!).

I cannot promise you the deepest coverage of the theory that goes into the database design process, nor do I want to. If you want to go to the next level, the latest edition of Chris Date's *An Introduction to Database Systems* (Addison Wesley) is essential reading, and you'll find hundreds of other database design books listed if you search for "database design" on a book seller's web site. The problem is that a lot 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. Other books that are implementation oriented don't give you enough theory and focus solely on the code and tuning aspects that one needs **after** the database is a mess. So many years ago, I set out to write the book you have in your hands, and this is the fifth edition under the Apress banner (with one earlier edition through a publisher to remain nameless). The technology has changed greatly, with the versions of SQL Server from 2012 and beyond ratcheting up the complexity tremendously.

WOW! eBook
www.wowebook.org

This book's goal is simply to be a technique-oriented book that starts out with "why" to design like the founders suggested, and then addresses "how" to make that happen using the features of SQL Server. I will cover many of 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 on the subject of database design, and particularly on SQL Server.

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 likely be writing this edition today. 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 for the third time. Both of them were very tough on my writing style and did wonders on the structure of the book (which is why this edition has no major structural changes). Another reason is simply experience, as over 15 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 (`www.drsql.org`). You may also find an addendum there that covers any material that I didn't have space for or 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.

WOW! eBook
www.wowebook.org

This process of designing and architecting the storage of data belongs to a different role than 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.

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 technology. 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 composed of the following chapters, with the first five chapters being an introduction to the fundamental topics and processes that one needs to go through/know before designing a database. Chapter 6 is an exercise in learning how a database is put together using scripts, and the rest of the book is takes topics of design and implementation and provides 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: Conceptual and Logical 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.

WOW! eBook
www.wowebook.org

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've 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: Expanding Data Protection with Check Constraints and Triggers. Beyond the way data is arranged in tables and columns, other business rules 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.

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 using other tools that are a part of the SQL Server toolset.

Chapter 10: Index Structures and Application. 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: Matters of Concurrency. As part of the code that's written, some consideration needs to be given to sharing resources, if applicable. 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 the databases you implement to provide a standard problem-solving interface for all of your systems while minimizing inter-database dependencies.

Chapter 13: Architecting Your System. This chapter covers the concepts and concerns of choosing the storage engine and writing code that accesses SQL Server. I cover on-disk or in-memory, 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), and whether T-SQL or CLR objects are best.

WOW! eBook
www.wowebook.org

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 downloadable appendix, including example tests of how they work and tips and pointers for writing effective triggers. (Appendix B is available as a download along with the code from www.apress.com or my web site.)

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 experience programming using any language.

Downloading the Code

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

Contacting the Author

Don't hesitate to give me feedback on the book, anytime, at my web site (www.drsql.org) or my e-mail (louis@drsql.org). I'll try to improve any sections that people find lacking in one of my blogs, or as articles, with links from my web site, currently at (www.drsql.org/Pages/ProSQLServerDatabaseDesign.aspx), but if that direct link changes, this book will feature prominently on my web site one way or another. 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.

WOW! eBook
www.wowebook.org

CHAPTER 1

The Fundamentals

Success is neither magical nor mysterious. Success is the natural consequence of consistently applying the basic fundamentals.

—Jim Rohn, American entrepreneur and motivational speaker

I have a love–hate relationship with fundamentals. The easier the task seems to be, the less enjoyable I seem to find it, at least unless I already have a love for the topic at 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 limited understanding of what makes one "good." Administrative assistants build databases using Excel, kids make inventories of their video games on a sheet of paper, and newbie programmers build databases with all sorts of database management tools, and rarely are any of the designs and implementations 100% wrong. The problem is that in almost every case the design produced is fundamentally flawed, causing future modifications to be very painful. When you are finished with this book, you should be able to design databases that reduce the effects of many of the 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: implementing a relational database. 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 PASS Summit (`www.sqlpass.org`).

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-1973-7_1) contains supplementary material, which is available to authorized users.

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_1

WOW! eBook
www.wowebook.org

For this chapter, we will start out looking at basic background topics that are very useful.

- *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 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.
- *Relationships between entities*: We will briefly survey the different types of relationships that can exist between 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 procedural 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. Throughout my years of reading and research, I've noticed that lack of agreed-upon terminology is one of the biggest issues in the database community. Academics have one (well, ten) set(s) of terms to mean the same thing as we people who actually develop code. Sometimes multiple different words are used to mean one concept, but worst case, one word means multiple things. Tradespeople (like myself, and probably you the reader) have their own terminology, and it is usually used very sloppily. I am not immune to sloppy terminology myself when chatting about databases, but in this book I do my best to stick to a single set of terms. Some might say that this is all semantics, and 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.

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 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.

WOW! eBook
www.wowebook.org

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 was 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 computer subsystems, 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 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 considered relational.

Introducing Codd's Rules for an RDBMS

I feel it is useful to start with Codd's rules, because while these rules are over 30 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 not only was 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.*

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 of rows and columns of data that users worked with.

Many different data structures were in use in the early days 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 worked just like everyone else's? Very few, and new innovations are coming every day.

WOW! eBook
www.wowebook.org

While innovation is rarely a bad thing, innovation in relational databases is largely 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 great advances in the language we use (SQL), but other than a few wonky bits of syntax that have been deprecated (the most common example being *= for left join, and =* for right [no *=* for full outer join]), SQL written 20 years ago will work today, largely because data is stored in structures that appear to the user to be exactly the same as they did in SQL Server 1.0 even though the internals are vastly different.

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 more depth 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 container; 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 the 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.*

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 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, 1 + "Not sure" does not compute.

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

WOW! eBook
www.wowebook.org

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. NULLs are a concept that academics have tried to eliminate as a need for years and years, but no practical replacement has been created. Consider them painful but generally 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 using the same tools that you store user data in. 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 well 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 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.

WOW! eBook
www.wowebook.org

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 from a table. This rule is really quite hard to implement in practice because views can be defined in any way the user wants.

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, as you had to manipulate data byte by byte, and usually you were just doing it for a single user at a time. 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.*

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. That way the folks that write the code for SQL Server's engine can add new fun features to the product and we can use many of them without even knowing (or at least, barely knowing) about them. For all we know, while the output of `SELECT * FROM <tablename>` would be the same in any version of SQL Server, the underlying code can be quite different

WOW! eBook
www.wowebook.org

(tremendously different when we look at how the new memory-optimized features will affect the internals and query processing!).

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 is concerned with the internal data structures that interface the relational engine to the file system, this rule is more centered on 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.

This principle, unlike physical data independence, does involve following solid programming practices. For example, consider the construct known as star (*) that is used as a wildcard for all of the columns in the table (as in `SELECT * FROM <tablename>`). Using this shorthand means that if a column is added to the table, the results will change in a way that might not be desirable. There are other places where this can cause issues (like using a column list in an `INSERT` statement), which we will cover throughout the book. Generally speaking though, it is always a good idea to declare exactly the data you need for any operation that you expect to 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 is that data should have integrity; and in this case that the data subsystem should be able to protect itself from most common 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.

Making complete use of the relational engine's integrity constraints can be controversial. Application programmers don't like to give up control of the management of rules because managing the general rules in a project must be done in multiple places (for user friendliness if for no other reason). At the same time, many types of constraints for which you need to use the engine are infeasible to implement in the application layer due to the desire to allow concurrent access. For example, uniqueness and referential integrity are extremely hard to implement from a client tool for reasons that probably are obvious in some respects, but will be covered in some detail in Chapter 11.

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.

WOW! eBook
www.wowebook.org

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. With cloud computing exploding considerably since the prior edition of this book came out in 2012, we are just getting closer and closer to a reality.

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 methods of accessing the data are not able to bypass everything that the relational engine has been specified to provide in the other rule, which means that users should not be able to 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 works on data in a manner that encapsulates the nasty technical details from the user. To fulfill this need, 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 (though we still regularly pronounce it as "sequel" today). 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 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

WOW! eBook
www.wowebook.org

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.

Obviously, there is a lot more history between 1985 and today. Many academics, including Codd himself, have advanced the science of relational databases to the level we have now. Notable contributors include C. J. Date, Fabian Pascal (who has a website `www.dbdebunk.com`), Donald Chamberlin, and Raymond Boyce (who contributed to one of the Normal Forms, covered in Chapter 6), among many others. Some of their material is interesting only to academics, but it all 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 all the other database design materials 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, without dumbing it down, 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 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 problem). 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.

WOW! eBook
www.wowebook.org

Introducing Databases and Schemas

A database at its core 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 many 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 a database server can be addressed by knowing the database they reside in and the schema, giving you what is known as the four-part name:

```
serverName.databaseName.schemaName.objectName
```

The only part of a name that is always required is the objectName, but as we will see later, always including the schema name is generally desirable. Including the database name and server name typically is frowned upon in normal use. These naming parts are generally acquired by the context the user is in, to make code more portable. A three-part name would be used to access a resource outside of the database in context, and a four-part name to access a database resource that is on another server. A goal in database development is to keep your code isolated in a single database if at all possible. Accessing a database on a different server is a practice disfavored by almost anyone who does database coding, first because it can be terrible for performance, and second because it creates dependencies that are difficult to track.

The database functions as 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.

Schemas are a layer of containership that you can use to segregate objects of like types or purpose. They are valuable not only for logical organization, but also, as we will see later, to control access to the data and restrict permissions. In Chapter 9, we will discuss in some detail the methods, values, and problems of managing security of data in separate databases and schemas.

■ **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

In a relational database, a table is used to represent some concept (generally a noun like a person, place, thing, or idea) and information about that concept (the name, address, descriptions, etc.). Getting the definitions of your tables correct is the most important part of database design, and something we will discuss in more depth.

WOW! eBook
www.wowebook.org

A table is the definition of the container for the concept. So, for instance, a table may represent a person. Each instance of a person "Fred Smith", "Alice Smith" is represented in "rows" of data. So in this table of people, one row would represent one person. Rows are further divided into columns that contain a single piece of information about whatever the row is representing. For example, the first name column of a row would contain "Fred" or "Alice." A table is not to be thought of as having any order and should not be thought of as a location in some type of 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.

"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 your neighbors will not appreciate the massive crater where your house previously was).

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 scalar value defined as two independent values, say `X` and `Y`, is perfectly acceptable because they are not independent of one another, while values like `'Cindy,Leo,John'` would likely not be atomic, because that value can be broken down into three separate values without losing any meaning. While you may be thinking that any programmer worth the price of a biscuit can split those values into three when they need to, our goal throughout the database design process is to do that work up front to provide the relational engine a consistent way of working with our data.

Before moving on, I would like to take a moment to discuss 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 (`www.dictionary.com`) has the following definition:

> *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.

WOW! eBook
www.wowebook.org

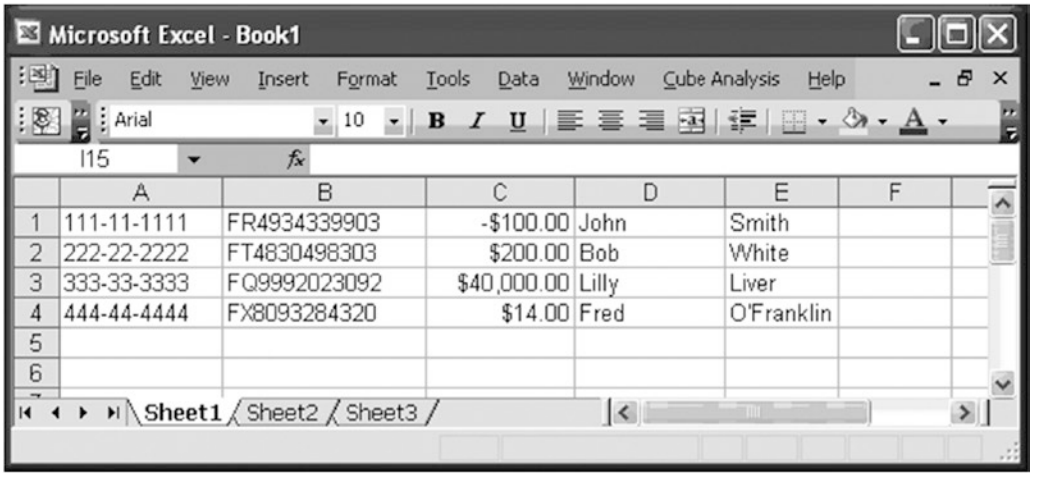


Figure 1-1. *Excel table*

In Figure 1-1, the rows are numbered 1–4, and the columns are labeled A–E. 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–A4) with no reference to the data's structure, something I have already mentioned several times as being against the principals of relational databases.

In the next few tables (in the book text—see, this term has lots of meanings!), 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 (note that there are quite a few other terms that mean the same things too, but these are the most common that I see in mainstream discussions):

- *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. Basically this is based on the concepts of Entity-Relationship (ER) modeling, which uses terms that are more generic than you will use when working with your database.
- *Physical*: This set of terms is used for the implemented database. The word "physical" is a 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 required 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.

WOW! eBook
www.wowebook.org

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. It is a strictly defined structure that is made up of tuples and attributes (which will be defined in greater detail later, but share a lot in common conceptually with rows and 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 represents a container for some "thing" you want to store data about. For example, if you are modeling a human resources application, you might have an entity for Employees. During the logical modeling phase, many entities will be identified, some of which will actually become tables and some will become several tables, based on the process known as normalization, which we'll cover extensively in Chapter 6. It should also be clear that an entity is not something that has an implementation, but is a specification tool that will lead us to an implementation.
Physical	Table	A table is, at its heart, very similar to a relation and entity in that the goal is to represent some concept that you will store data about. The biggest difference between relations and tables is that tables technically may have duplication of data (even though they should not be allowed to in our implementations). If you know about views, you may wonder how they fit in. A view is considered a type of table, often a "virtual" table, since it has a structure that is considered permanent.
Record manager	File	In many nonrelational-based database systems (such as Microsoft FoxPro), each operating system file represents a table (and sometimes a table is actually referred to as a database, which is just way too confusing). Multiple files make up a database. SQL Server employs files, but at the engine level and may contain all or parts of one or more tables. As previously mentioned regarding Codd's rules, how the data is stored should be unimportant to how it is used or what it is called.
Physical	Recordset/ rowset	A recordset, or rowset, is data that has been retrieved for use, such as 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 have some similarity to a normal table, but their differences are tremendous as well. Seldom will you deal with recordsets in the context of database design, but you will once you start writing SQL statements. A major difference between relations/tables and recordsets is that the former are considered "sets," which have no order, while recordsets are physical constructs used for communication.

WOW! eBook
www.wowebook.org

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.

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 modeling, this term can be applied to almost anything, and it may actually represent other entities. Just as with entities, in order to produce proper tables for implementation, normalization will change the shape of attributes until they are proper column material.
Relational theory	Attribute	When used in relational theory, an attribute takes on a strict meaning of a scalar value that describes the essence of what the relation is modeling.
Physical	Column	A column is a single piece of information describing what a row represents. The position of a column within a table is strongly suggested to 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. Each column value is expected to be, but isn't strictly enforced, an atomic scalar value.
Record manager	Field	The term "field" has a couple of meanings in a database design context. 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.

■ **Note** 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. Additionally, some support for translating and reading JSON-formatted values has been added to SQL Server 2016, though there is no formal datatype support.

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

WOW! eBook
www.wowebook.org

Table 1-3. *Row Term Breakdown*

Viewpoint	Name	Definition
Relational theory	Tuple	A tuple (pronounced "tupple," not "toople") is a finite unordered set of related, and also unordered set of named value pairs, as in ColumnName: Value. 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 database 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. Note that tuple is used in cubes and MDX to mean pretty much the same concept, if things weren't confusing enough already. An important part of the definition of a relation is that no two tuples can be the same. This concept of uniqueness is a topic that is repeated frequently throughout the book because, while uniqueness is not a strict rule in the implementation of a table, it is a very strongly desired design characteristic.
Logical/ conceptual	Instance	Basically, as you are designing, you will think about what one version of an entity would look like, and this existence is generally referred to as an instance.
Physical	Row	A row is essentially the same as a tuple, with each column representing one piece of data in the row that represents one thing that the table has been modeled to represent.
Record Manager	Record	A record is considered to be a location in a file on disk. Each record consists of fields, which all have physical locations. Ideally, this term should not be used interchangeably with the term "row" because a row is not a physical location, but rather a structure that is accessed using data.

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 is 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 "record" when they really mean "row" or "tuple," but I also realize that misusing a term isn't the worst thing if a person understands everything about how a table should be dealt with in SQL.

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 means. Since the value is missing, it is assumed that there may exist a value (even if that value is that there is specifically no value that makes sense.) Because of this, no two values of `NULL` are considered to be equal, and you have to treat

WOW! eBook
www.wowebook.org

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 evaluate to NULL, for the very same reason that any value +/*- or any mathematical equation with an unknown value will be unknown (even 0 * NULL evaluates to NULL).
- Logical comparisons can get tricky when NULL is introduced because NULL <> NULL (the resulting Boolean expression is NULL, not FALSE, since any unknown value might be equal to another unknown value, so it is unknown if they are not equal). Special care is required in your code to know if a conditional is looking for a TRUE or a non-FALSE (TRUE or NULL) condition. SQL constraints in particular look for a non-FALSE condition to satisfy their predicate.

Let's expand this point on logical comparisons somewhat, as it is very important to getting NULL usage correct. 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 a 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;
```

Since you have two conditions, and the second condition is the opposite of the first, it seems logical that either NOT(1=NULL) or (1=NULL) would 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 1-4. *NOT Truth Table*

Operand	NOT(Operand)
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

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

WOW! eBook
www.wowebook.org

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

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 eliminating them is very nearly impossible, particularly because they begin to appear in your SQL statements even when you do an outer join operation.

Defining Domains

All of the concepts we have discussed so far have one very important thing in common: they are established to help us end up with some structures that store information. Which information we will store is our next consideration, and this is why we need to define the *domain* of a structure as the set of valid values that is allowed to be stored. At the entity level, you will specify the domain of an entity by the definition of the object. For example, if you have a table of employees, each instance will represent an employee, not the parts that make up a ceiling fan.

For each attribute of an entity you specify, it is necessary to determine what data is allowed to be contained in the attribute. As you define the domain of an attribute, 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 an attribute 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 usually be less than 70 years ago, since rarely will an employee (especially a new employee) be that age.
- The value must be less than 130 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 70 years old. 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 "C'mon, really?" for values out of normal bounds).

WOW! eBook
www.wowebook.org

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 would be akin to reinventing sliced bread. To make this process easier and to achieve some standards among your models, a great 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 bottles of hot sauce. Puppies and hot sauce don't mix (only older dogs use hot sauce, naturally), but the domain of how many you have is very similar. 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; Rodney Landrum, a technical editor for the book has this as a tattoo, if you were wondering), so it is useful to define how many is too many (can you really have a billion puppies?). 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 I discuss data integrity, as well as in Chapter 8 when I discuss patterns of implementation to meet requirements.

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 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 attribute name/datatype name pairs that defines the attributes of the tuple
- *Body*: The tuple that makes up the relation

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

- *In a set of views known as the information schema*: It consists of a standard set of views used to view the system metadata table 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.

WOW! eBook
www.wowebook.org

It is a very good practice to maintain your own metadata about your databases to further define a table's or column's purpose than just naming objects. 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).

Defining 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 effectively retrieve a single row.

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

Consider the following set of data, named `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:

- 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. Using, changing, or deleting an individual row would be difficult without resorting to tricks with the SQL language (like using a `TOP` operator).
- 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, whereas a key with only one column is sometimes referred to as a "simple" key), 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.

■ **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.

WOW! eBook
www.wowebook.org

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, that need may not 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 persist until the real users start testing (using) the database.

In summary, a candidate key (or simply "key" for short) defines the uniqueness of rows over a column or set of columns. In your logical designs, it will be essential for each entity to have one uniqueness criteria set, and it may have multiple keys to maintain the uniqueness of the data in the resulting tables and rows, and a key may have as many attributes 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 in more detail implementation patterns for implementing uniqueness conditions of several types 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 numbers, 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.

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.

WOW! eBook
www.wowebook.org

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 entity with the attributes Publisher_Name, Publisher_City, ISBN_Number, Book_Name, and Edition. From these values, 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 (International Standard Book 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, Edition) 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 since we have already said it must be unique on its own. 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** Unique indexes should not be confused with uniqueness keys. There may be valid performance-based reasons to implement the Publisher_City, ISBN_Number unique index in your SQL Server database. However, this would not be identified as a key of a entity during design. 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 key*: The values that make up the key have some connection to the row data outside of the database context
    - *Smart key*: A type of natural key that uses a code value to pack multiple pieces of data into a short format.
- *Surrogate key*: Usually a database-generated value that 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 based on a relationship to the entity that exists outside of the database. 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. Of course, a number like a U.S. SSN doesn't look natural because it is 11 digits of numbers and dashes, but it is considered so because it originated outside of the database.

WOW! eBook
www.wowebook.org

Natural keys are values that a user would recognize and would logically be presented to the user. Some common examples of natural keys are

- *For people*: Driver's license numbers + state of issue, company identification number, customer number, employee number, etc.
- *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), UPC code
- *For buildings*: A complete street address, including the postal code, GPS coordinates
- *For mail*: The addressee's name and address + 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 can be 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 possibly actually more of a smart key, or possibly a surrogate key (which we will define in a later section).

Values for which you cannot guarantee uniqueness, no matter how unlikely the case, should not be considered as 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 (I know I work with three people, two of them fellas, with that name, because who doesn't, right?), 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, might do just that).

One notable profession where names must be unique is actors. 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 (`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!)

WOW! eBook
www.wowebook.org

■ **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 an entity and should be documented for later use, but only in rare cases would you make a key from semiunique values, probably by adding an uniquifier. In Chapter 8, we will cover the process of defining and implementing this case, which I refer to as "likely uniqueness." Likely uniqueness criteria basically states that you should ask for verification if you try to create two 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, which I have devised to be broken down into the following parts:

- *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 almost certainly going to end up stored in its own column.

Smart keys, while useful as a human value that condenses a lot of information into a small location, definitely do not meet the criteria we originally set up as scalar earlier, and by the time when we start to implement database objects certainly should not be the only representation of these pieces of data, but rather implementing each bit of data as a single column with each of these values, and determine how best to make sure it matches the smart key if it is unavoidable.

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.

WOW! eBook
www.wowebook.org

■ **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.

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, or is perhaps too unwieldy to work with, in your model.

A surrogate key can give you a unique value for each 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, the modeler, or the client application, or a combination of them. Common methods for creating surrogate key values are to use a monotonically increasing number, to use a random value, or even to use a globally unique identifier (GUID), which is a very long (16-byte) identifier that is unique on all machines in the world.

The concept of a surrogate key can be troubling to purists and may start an argument or two. Since the surrogate key is not really information, can it really be an attribute of the entity? The question is valid, but surrogate keys have a number of nice values for usage that make implementation easier. For example, an exceptionally nice aspect of a surrogate key is that the value of the key need 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 the series of articles at `www.highprogrammer.com/alan/numbers/index.html` 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).

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

WOW! eBook
www.wowebook.org

- *9*: Last digit of year
- *2*: Original color
- *000123*: Order of production

A natural key would almost certainly consist of these seven parts since this is smart key (it is possible that a subset of the values forms the key with added data, but we will assume all parts are needed for this example). 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—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 values 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.

■ **Note** When doing early design, I tend to model each entity with a surrogate primary key, since during the design process I may not yet know what the final keys will turn out to be until far later in the design process. 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 an entity is and how entities are to be structured (especially with an eye on the future tables you will create), but an entity by itself can be a bit boring. To make entities 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 an entity to itself). Without the concept of a relationship, it would often 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 a primary no-no in good database design).

A term alluded to earlier that we need to establish is "*foreign key*." A foreign key is used to establish a link between two entities/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 is generally allowed, with some caveats having to do with NULLs that we will cover when we get to implementation of the concepts).

WOW! eBook
www.wowebook.org

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

- *Involvement*: The entities 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 entities need not be two. Sometimes, it is just one, such as an employee entity 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 common entities that would be related to one another in a complex relationship.
- *Ownership*: It is common that one entity will "own" the other entity. 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!).

When we begin to implement tables, there will be a limitation that every relationship can only be between two tables. The relationship is established by taking the primary key columns and placing them in a different table (sometimes referred to as a "migrated key"). 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. (Note that the two tables in the relationship may be the same table, playing two roles, like an employee table where the relationship is from manager to managee, as we will see later.)

For an example of a relationship between two tables with some representative data, 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, this is a simple example that does not take into full consideration all of the intricacies of people's names.

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!).

Cardinality is the next question. It is important when defining a relationship to know how many parent rows can relate to how many child rows. Based on the fact that the `Parent` entity has its key migrated to the `Child` entity, we have the following restriction: one parent can have any number of children, even zero,

WOW! eBook
www.wowebook.org

based on whether the relationship is considered optional (which we will get to later in the chapter). Of course, if this were really a parent–child human relationship, we could not limit parents to one, because every human (still) has two biological parents, and the number of persons considering themselves parents is even more varied. What makes the job of database design challenging and interesting is all of the realities of the world that don't always fit into nice neat molds.

Relationships can be divided at this point into two basic types based on the number of entities involved in the relationship:

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

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 entities are involved, we will generally use methods of modeling and implementation to break down the design into a series of binary relationships, without losing fidelity or readability of the original relationship that your customer will expect.

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 3, you'll learn how to represent relationships of many types in a structured representation.

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 relationship*: A relationship linking one instance of an entity with (possibly) multiple instances of another entity. This is the type of relationship that we will implement in our database's tables.
- *Many-to-many relationship*: Generally, a relationship where one instance in one entity can be linked with multiple instances of the second entity, and instances in the second entity can in turn be related to many instances in the first one. This is the most typical relationship type to occur in reality.

While all relationships you physically implement in your code will be one-to-many, the real world is not so clear. Consider a football player's relationship to teams. One team has many players, and a player can play on one team. But a player over their career can play for many teams. So the player-to-team relationship is also a many-to-many relationship. Parent to child, as we previously discussed, is a many-to-many relationship because two biological parents may have many children.

We will look at 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 entity migrates its primary key to another 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 entities. A child may have, at most, one parent, but a parent may have one or more than one child instances. 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 number, or an unlimited number.

WOW! eBook
www.wowebook.org

It should be immediately clear that when the type of relationship starts with "one-to-," such as "one-to-many," 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), as shown in Figure 1-3.

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*

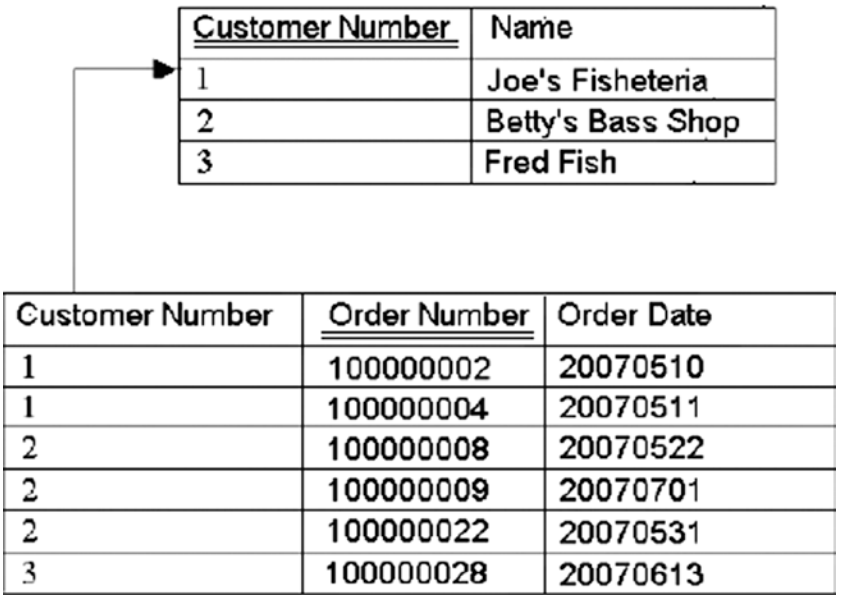
The missing value would typically be denoted by `NULL`, so the row for Sonny would be stored as (`ChildSSN:'666-66-6666', ChildName:'Sonny', ParentSSN:NULL`). For the general case, we (and most others in normal conversation) will speak in terms of one-to-many relationships, just for ease of discussion. However, in more technical terms, there are several different variations of the zero or 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 must have two biological parents, so it would be one-to-exactly 2 (though discussion about whether both parents must be known to record a child's existence is more of a topic for Chapter 2, when we discuss requirements. The common exact case is 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 between 1 and 2 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.

WOW! eBook
www.wowebook.org



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*

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 entity, and often the relationship is set up as a single entity. This kind of relationship is used to model a tree data structure. 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 of each of its parts, and in turn, each of those parts consists of other parts, and so on. (Note that a bill of materials is a slice of a larger "graph" structure to document inventory, as each part can have multiple uses, and multiple parts. This structure type will be covered in more detail in Chapter 8.)

The following table is a 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.

```
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
```

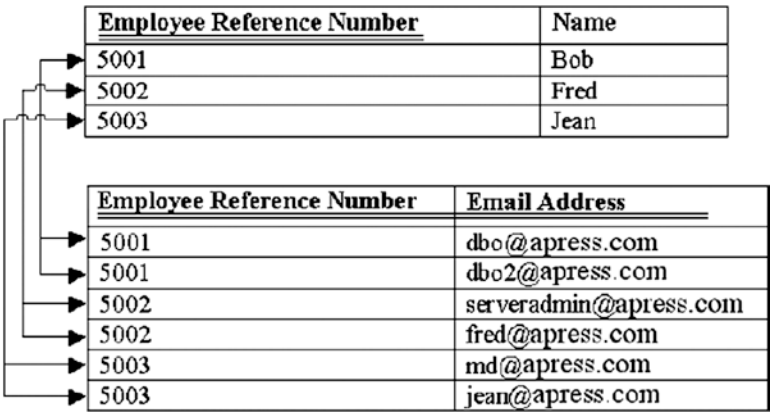
To read this data, you would start at `Part Number 1`, and you can see what parts make up that part, which is a fan blade kit and a light assembly. Now, you have the parts with number 2 and 3, and you can look for parts that make them up, which gets you part number 4 and 5. (Note that the algorithm we 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. It's not terribly important at this point, but it will come up in Chapter 8 when we are discussing design patterns.)

WOW! eBook
www.wowebook.org

One–to–N Relationship

Often, some limit to the number of children is required by the situation being modeled or a business rule. So rather than one-to-many (where many is infinity), you may say we only support a specific number of related items. A person playing poker is dealt five cards. Throughout the game the player has between zero and five cards, but never more.

As another example, a business rule might state that a user must have exactly two e-mail addresses so they can be more likely to answer one of the e-mails. Figure 1-5 shows an example of that one-to-exactly two relationship cardinality. It's not particularly a likely occurrence to have data like this, but you never know whether you need a tool until a customer comes in with some wacky request that you need to fill.



Employee Reference Number	Name
5001	Bob
5002	Fred
5003	Jean

Employee Reference Number	Email Address
5001	dbo@apress.com
5001	dbo2@apress.com
5002	serveradmin@apress.com
5002	fred@apress.com
5003	md@apress.com
5003	jean@apress.com

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

The most typical version of a one–to–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. A one-to-one relationship may be a simple attribute relationship (e.g., 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 is an extension of 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. These types of "is a" relationships are often what are called subtype relationships, something we will cover again later in the book.

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, you have two entities where each instance in both entities can be tied to any number of instances in the other. For example, continuing with the familial relationships, a child always has two biological parents—perhaps unknown, but they do exist. This mother and father may have more than one child, and each mother and father can have children from other relationships as well.

WOW! eBook
www.wowebook.org

Many-to-many relationships make up a lot more of what we will model than you might immediately expect. Consider a car dealer. The car dealer sells cars. Inversely, pick nearly any single model of car, and you'll see that it is sold by many different car dealers. Similarly, the car dealer sells many different car models. So many car models are sold by many car dealers, and vice versa.

As another 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 going to be 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 structure to implement the relationship. Instead of the key from one entity being migrated to the other, the keys from both objects in the relationship are migrated to a new entity that is used to implement the relationship (and likely record information about the nature of the relationship as well). 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 entities 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 entities 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 entity to associate them all into one relationship:

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

We now have an entity that seemingly 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 entities (commonly known as a ternary relationship, because of the three entities) 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 series of binary relationships that are easy to work with. This is part of the normalization process that will be covered in Chapter 5 that will change our initial entities into something that can be implemented as tables in SQL. During the early, conceptual phases of design, it is enough to simply locate the existence of the different types of relationships.

WOW! eBook
www.wowebook.org

Understanding Functional Dependencies

Beyond basic database concepts, I want to introduce a few mathematical concepts now before they become necessary later. They center on the concept of functional dependencies. The structure of a database is based on the idea that given one value (as defined earlier, a key 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). The 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 is the person's eye color?"

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.

Understanding Functional Dependencies

Functional dependency is a very simple but important concept. It basically means that if you can determine the value of variable A given a value of variable 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`. Then if you are certain that for every input to the function (`Value1-1`, `Value1-2`, ..., `Value 1-N`) that you will always get back (`Value2-1`, `Value2-2`, ..., `Value 2-N`), the function that changes `Value1` to `Value2` is considered "deterministic." On the other hand, if the value from the function can vary for each execution, it is "nondeterministic." This concept is central to how we form a database to meet a customer's needs, along with the needs of an RDBMS engine.

In a table form, 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 (note that these functions are not strictly math; it could be `IF X = (2 or 3) THEN 2 ELSE 1` that forms the function in question). 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
```


WOW! eBook
www.wowebook.org

Determining what is a dependency and what is a coincidence is something to be careful of as well. In this example, `fn(X) = Y` and `fn(X) = Y` for certain (since that is the definition of a key), but looking at the data, there also appears to exist another dependency in this small subset of data, `fn(Y) = Z`. Consider that `fn(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 `fn(Y) = Z` because `fn(2) = 5 AND fn(2) = 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 with just a little bit of planning.

Finding 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.

If this all seems familiar, it is because any key of a entity will functionally determine the other attributes of the entity, and each key will be a determinant, since it functionally determines the attributes of the entity. If you have two keys, such as the primary key and alternate key of the entity, each must 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, per 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, authorization, 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 on using SQL to do all of the things listed in the fifth rule and more.

WOW! eBook
www.wowebook.org

There are two characteristics of SQL that are particularly important to understand. First, SQL is at its core a declarative language, which basically means that the goal is to describe what you want done to the computer, and SQL works out the details.[1] So when you make the request "Give me all of the data about people," you ask SQL in a structured version of that very sentence, rather than describe each individual step in the process.

Second, SQL is a relational language, in that you work at the relation (or table) level on sets of data at a time, rather than on one piece of data 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 declarative statement almost always represents hundreds and thousands of lines of code being executed. Much of this code executes in the hardware realm, accessing data on disk drives, moving that data into registers, and performing operations in the CPU.

If you are already well versed as a programmer in a procedural language like C#, FORTRAN, VB.NET, etc., SQL is a lot more restrictive in what you can do. You have two sorts of high-level commands:

- *Data Definition Language (DDL)*: SQL statements used to set up data storage (tables and the underlying storage), 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. Even more important is as a relational data designer, it is your job to make sure the database suits the needs of the RDBMS. Think of it like a chef in a kitchen. Producing wonderful food is what they do best, but if you arranged their utensils in a random order, it would take a lot longer to make dinner for you.

Dr. David DeWitt (a technical fellow in the Data and Storage Platform Division at Microsoft Corporation) said, during his PASS Keynote in 2010, that getting the RDBMS to optimize the queries you send isn't rocket science; it is far more difficult than that, mostly because people throw ugly queries at the engine and expect perfection. Consider it like the controls on your car. To go to the grocery store and get milk, you don't need to know about the internal combustion engine or how electric engines work. You turn the car on, put it in gear, press pedal, and go.

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 phases of a database project 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

[1]For a nice overview of declarative languages, see https://www.britannica.com/technology/declarative-language.

WOW! eBook
www.wowebook.org

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. Next, think about how to implement using proper fundamental techniques, and finally, tune the implementation to work in the real world.

The process I outline here steers us through creating a database by keeping the process focused on getting things done right, using terminology that is reasonably common in the general practicing programmer community:

- *Conceptual*: During this phase, the goal is a sketch of the database that you will get from initial requirements gathering and customer information. I said Phase, you identify what the user wants at a high level. You then capture this information in a data model consisting of high-level entities and the relationships between them. (The name "conceptual" is based in finding concepts, not a design that is conceptual. Think of them like storyboards of a movie. They storyboard to make sure everything seems to work together, and if not, changes require an eraser, not completely reshooting a film.)
- *Logical*: The logical phase is an implementation-nonspecific refinement of the work done in the conceptual phase, transforming the concepts into a full-fledged relational database design that will be the foundation for the implementation design. During this stage, you flesh out the model that the system needs and capture all of the data business rules that will need to be implemented. (Following the movie analogy, this is where the "final" script is written, actors are hired, etc.)
- *Physical*: In this phase, you adapt the logical model for implementation to the host RDBMS, in our case, SQL Server. For the most part, the focus in this phase is to build a solution to the design that matches the logical phase output. However, a big change has occurred in recent years, and that is based on SQL Server internals. We will need to choose between a few models of implementation during the physical phase. (The physical database is like the movie now in the can, ready to be shown in theaters. The script may change a bit, but overall it will be very close.)
- *Engine adjustment*: In this phase, you work with the model where the implementation data structures are mapped to storage devices. This phase is also more or less the performance tuning/optimization/adaption 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/client software looks like. It might not function very fast, but it will function. During this phase of the project indexes, disk layouts, are adjusted to make the system run faster, without changing the meaning of the system. (The engine adjustment phase of the project is like distributing a film out to millions of people to watch on different screens, different sized theaters, formats, etc. It is the same film on a massive screen as on your phone screen, with optimizations made so it works in each situation.)

You may be questioning at this point, "What about testing? Shouldn't there be a testing phase?" Yes, each of the phases includes testing. You test the conceptual design to make sure it fits the requirements. You test the logical design to make sure you have covered every data point that the user required. You test the physical design by building code and unit and scenario testing it. You test how your code works in the engine by throwing more data at the database than you will ever see in production and determining whether the engine can handle it as is or requires adjustments. A specific testing phase is still necessary for the overall application, but each of the database design/implementation phases should have testing internally to the process (something that will equally be sewn into the rest of the book as we cover the entire process).

WOW! eBook
www.wowebook.org

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).

Over the years, I have 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 a diagram of entities and relationships. Others included attributes and keys as they were describing it.

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 between if possible them, the goal being to capture, at a high level, the fundamental concepts 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 I have worked with have almost always documented as much of the information as they can in the model as they find it. I personally find the discipline of limiting the model to entities and relationships to be an important first step, and invariably the closer I get to a correct conceptual model, the shorter the rest of the process is.

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 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.
- Identifying the attributes (or a group of attributes) that make up candidate keys. 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 including 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).

WOW! eBook
www.wowebook.org

■ **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.

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 validating the design (using the rules of what is known as normalization, covered in Chapter 5), then choosing datatypes for columns to match the domains, building tables, applying constraints and occasionally writing triggers to implement business rules, and so on to implement the logical model in the most efficient manner. This is where reasonably deep 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, concurrency needs, 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 points can (and will) be added, often to handle the process of writing programs to use the data, like data to know who created or was the last person to change a row. 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 (though to be clear, we may design security last, we shouldn't design it after you finish coding, in case it affects the design in some way).

Engine Adjustment Phase

The goal of the storage layout phase is to optimize how your design interacts with the relational engine—for example, by implementing effective data distribution on the physical disk storage and by judicious use of indexes, or perhaps changing to use the columnstore or in-memory structures that Microsoft has implemented in the past few years (and has significantly improved in SQL Server 2016 with columnstore indexes that can be used for reporting on OLTP tables without blocking writers, editable columnstore indexes, and other new features). 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.

As more and more data is piled into your database, ideally the only change you will need to make to your database is in this phase (assuming the requirements never change, which is reasonably unlikely for most people). This task is very often one for the production DBA, who is charged with maintaining the database environment. This is particularly true when the database system has been built by a third party, like in a packaged project or by a consultant.

WOW! eBook
www.wowebook.org

■ **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 the latest in the Internals series by Kalen Delaney (`http://sqlserverinternals.com/`). To see where the industry is rapidly 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 throughout this book. I pretty much guarantee that the need for a natural key on every table (in the resulting database that is obviously your goal for reading this far in the book) 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 to you).

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 `NULL`s and relationships. We also looked at a basic introduction to the impetus for how SQL works.

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 5 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 what our data ought to look like, we'll shift gears to the implementation phase and speak of tables, rows, and columns. The terminology is not terribly important, and in the real world it is best not to be a terminology zealot, but when learning, it is a good practice to keep the differences distinct. The really exciting part comes as 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.

The rest of the book will cover the process with the following phases in mind (the next chapter will be a short coverage of the project phase that comes just before you start the conceptual design requirements):

- *Conceptual*: Identify the concepts that the users expect to get out of the database system that you are starting to build.
- *Logical*: Document everything that the user will need in their database, including data, predicates/rules, etc.

WOW! eBook
www.wowebook.org

- *Physical:* Design and implement the database in terms of the tools used (in the case of this book, SQL Server), making adjustments based on the realities of the current version of SQL Server/other RDBMS you are working with.
- *Engine adjustment*: Design and lay out the data on storage based on usage patterns and what works best for SQL Server. Adjust the design such that it works well with the engine layers, both algorithms and storage layers. The changes made 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 I have been told.

WOW! eBook
www.wowebook.org

CHAPTER 2

Introduction to Requirements

Being busy does not always mean real work. The object of all work is production or accomplishment and to either of these ends there must be forethought, system, planning, intelligence, and honest purpose, as well as perspiration. Seeming to do is not doing.

—Thomas Edison, American Inventor and Businessman

If there is anything worse than doing a simple task without fully understanding the requirements for success, it is doing a complex one. It happens every day, computer projects are created with only a shimmer of an idea of what the end goal is. Sometimes this is really interesting, and leads you to something interesting. . . sometimes, but almost never when the goal of that project is to deliver value to a customer who knows what they want. Hence, *someone* involved in every software project must take time to figure out what the customer wants before any software designing starts. If you are very lucky, this will not be you, as capturing requirements is considerably more difficult than any task that will follow in later chapters.

However, for the rest of this chapter, we are going to assume that we have been given a business analyst hat to wear and we need to gather the requirements before we start to design a database. We will keep it very simple, covering only the minimal amount that can be gotten away with.

The first thing one generally does when starting a computer project is to interview users and ask a question along the lines of "Just what is it that you want?" And then listen, being mindful that users often aren't technologists. Some of them are. Still others completely believe they are and are not. So be sure and get lots of people involved and truly take the time to understand the customer's problems before you try to solve them. A major issue from my own history was that I would try to rapidly get past understanding the customer needs and right into design of the internals of a database.

The problem is that users don't think about databases; they think about user interfaces (UIs) and reports and how they want things to look and feel. 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 getting the right kinds of requirements to begin the database design process from your business analyst.

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 designing (and coding). No matter how you slice it, planning every 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,

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_2

WOW! eBook
www.wowebook.org

the importance of making sure someone gathers requirements cannot be understated. During a software project (and really any project, but let's focus on software projects), the following phases are common:

- *Requirements 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 requirements 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 requirements is like taking a trip without a 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 user's 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 name of "waterfall method" because the output of one step flowed into another. Waterfall has been derided as generally terrible (certainly uncool), because each phase is done to completion before moving to the next. More and more teams are using "agile" project development methods, but the phases are the same as in the waterfall method, the major differences being in the iterations and time slices. You still need to know what you are doing before you design, and design before you code.

The important point I want to make clear in this chapter is simple: each of these phases 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 (although there is a section about data quality and testing in Chapter 13). However, after this chapter, I am going to assume requirements are finished, and the design phase has begun in earnest. 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 a deep 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 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 Texas Instruments TI3243 calculator." A calculator might be the solution, and that may be a great one, but the decision should be left to the technologist to avoid overly limiting the final solution.

WOW! eBook
www.wowebook.org

- The language used should be as 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, in title only, and after six months, change his designation to DBA. (Users will find a way to get their job done!) 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. Simple diagrams and pictures also work nicely as communication devices.

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. Later, you will use requirements to test your design and code to make sure you have met them.

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 if the system you create stinks because the requirements that you were given stink, it won't be the requirements gatherer who has 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 incomprehensible 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. This sounds really obvious to say, but 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. But that is what we often expect of our users.

WOW! eBook
www.wowebook.org

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. Sometimes people take vacations, departing with vast amounts of job knowledge that is in their head only, and sometimes they don't come back from vacation (you know, because they get offered a job as a Jungle Cruise captain at Disneyland!). Many variations on this scenario exist, not all pleasant to contemplate. Without documentation, you will quickly risk losing 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 end up staying in the Magic Kingdom).

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 do your best to scope the requirements the same. 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 ones that will increase the budget, early in the process will avoid future animosity.
- Be succinct, but thorough. A document can be 1000 pages or 10 and say the same thing. The big test: Can the users, architects, and programmers use it and produce the end goal?

Once you produce a document, a crucial step follows: make sure the client agrees with your version of their goals. 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, as they are ultimately in 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. 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 probably closer to reality.

In addition to talking to the client, it's important to acquire as many notes, printouts, screenshots, portable drives, etc. 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

WOW! eBook
www.wowebook.org

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.

■ **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

Gathering requirements can be a daunting task, so start small and keep adding until you and your customer agree that you understand the problem well enough to move past it. 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, an Agile approach, like Scrum, breaks down everything into small units for designing, coding, testing, and releasing frequently (for example, every two weeks), while something like the waterfall methodology would expect you to design the entire system first and then start coding. No matter what your methodology, the eventual outcome should at least resemble the same thing (with allowances for change in technology and corporate needs). The important thing is that all development methodologies will tell you one thing: design what you are going to code *before* you code.

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; the Microsoft Solutions Framework (which employs UML) and Rational Unified Process are others. There are also several model types in the IDEF family of methods for business process modeling as well; we will cover their 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 pet half-bee Eric to make good word choices when playing Words with Friends ("Buzz Again?").

Regardless of the tools used to document the requirements, the needs for the database design process are the same. Specifications need to be acquired that will lead to you as the database designer to discover all of 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).

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, which we will be covering in the following chapters.

WOW! eBook
www.wowebook.org

■ **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 activities that you will need to be very interested in as a database architect:

- Interviewing clients
- Asking the right questions
- Working with existing systems and prototypes

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 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 be not only 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 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 other layers of business and interface objects—is where all the main work takes place.

WOW! eBook
www.wowebook.org

■ **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 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 (and like anyone with lots of experience, I have made my fair share of mistakes in all levels of software engineering). 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

Before painting the interior of any 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? children's room? write on/wipe off use?). 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 project team, 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 groupings. It will be very important at some time in your process to differentiate between what data is needed and what would just be nice to have. It is obviously a really straightforward question, and the customer may have no idea what they need, but many users already work with data, either on existing systems (that they either hate and will be glad you are replacing for them; or will hate you for changing things) or in a spreadsheet system that they have been using since VisiCalc.

WOW! eBook
www.wowebook.org

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.

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.

WOW! eBook
www.wowebook.org

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, and they never tried to simulate this condition in their user acceptance testing, so teeth were gnashed and sleep was lost fixing the problem.

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 limited 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 computer system profitable is the ability to report on all of the activity in the system in great detail. How productive different parts of the organization are, how effective salespersons are (on the second Tuesday after the first Wednesday, because, well, just because every business does stuff that seems wacky to outsiders. . .and some insiders). All of these questions 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, and have suffered through many, many times). 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. Note that it isn't a tools problem. Reporting Services, Power BI, etc. all make the tasks easier. The problem lies in knowing what data to capture. If you need to know the temperature of a freezer over the last five days, a part of the initial design is to capture the temperature in a temporal manner, over at least five days. If you didn't know that requirement, you might just have a single value for the current freezer temperature and not be able to meet the reporting needs. That would be an easy problem to solve, but often there are many little annoying problems like this that come up late in the process that end up implemented in a less than awesome manner. Getting it right the first time is definitely better than not.

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.

WOW! eBook
www.wowebook.org

■ **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. Specific reporting architecture is covered in more detail in Chapter 14.

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 Browns winning the Super Bowl (no offense, Browns fans, but 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. (Several have been brand new systems. These were wonderful experiences 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.

WOW! eBook
www.wowebook.org

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, Oracle boxes, or Hadoop systems, 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

---

How Much Is This Data Worth?

It's 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 requirements 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 around the world 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.

WOW! eBook
www.wowebook.org

- *Concurrency*: The design considerations for a system that has one simultaneous user are different from those for a system that has ten, or a hundred, thousand, and so on. The number of users should not change our conceptual or logical designs, but it will certainly change how we design the physical layer. Concurrency is something we won't make a lot of reference to until very late in this book (Chapter 11), but this is the point in time when you are doing the asking and likely specifying the future hardware\software, 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.

Are Existing Systems Being Replaced?

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 move to newer technology, but no one replaces a working system just for kicks. The worst part of replacing an existing system? Testing. Old System says OrderCount: 100; New System says it is 102. Which is right? Someone will need to prove the new system is right, no matter how well the new code has been tested.

Another thing that makes existing systems complicated is exiting data. If you have 20 years of previous data, perhaps based on a proprietary hierarchical file–based mainframe application, two problems often arise. First: the new system "needs" to work like the old system. Second: 20 years of data often presents tons of structural issues. So part of the requirements, design, and plan for implementation needs to address how to get the old data in.

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.

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, I find it best to put into writing your understanding and make sure it is clear who said what about the meaning of documentation you cannot see. 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.

WOW! eBook
www.wowebook.org

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 maybe they will let you keep the Formula 1 car?).

Level of Service Agreement

One important section of contracts that's also important to the design process is the required level of service. This might specify the number of web 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.

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 agencies, ISO 9000 clients, and other clients that are monitored by standards organizations are likely to have strict audit requirements. Other clients may

WOW! eBook
www.wowebook.org

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 require not only 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.

Prototypes

A prototype is kind of a live document that gets created so that the user can get a feel for how software might work for them. Prototypes are fantastic communication devices, but they are focused on visuals, not internals. 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. As a data architect, you must work as hard as possible to use prototype code *only* as a working document that you use to inform 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.

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.

Summary

In this chapter, I've touched on 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 users' needs.

WOW! eBook
www.wowebook.org

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 all of 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

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.

WOW! eBook
www.wowebook.org

CHAPTER 3

The Language of Data Modeling

I prefer drawing to talking. Drawing is faster, and leaves less room for lies.

—Le Corbusier, Swiss-French architect, designer, painter, urban planner

A data model is one of the most important tools in the design process, but it has to be done right. It very often starts as a sketch of the data requirements that you use to communicate with the customer, and is refined over and over until you get it right. However, a common misconception is that a data model is only a picture of a database. That is partly true, but a model is truly so much more. A great data model includes nongraphical representations of 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 of the 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 user interface and overview of your objects and their relationships).

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 parts of the database (introduced in the prior two chapters):

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

In the process of creating a database, we will start out modeling entities and attributes, which will start fairly loosely defined until we start to formalize tables and columns, which, as discussed in Chapter 1, have very formal definitions that we will continue to refine in Chapter 5. For this chapter and the next, I will primarily refer to entities during the modeling exercises, unless I'm trying to demonstrate something that would specifically 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.

After introducing IDEF1X, I 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 will also briefly mention some of the tools that SQL Server and Office provide

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_3

WOW! eBook
www.wowebook.org

for viewing a database model, though generally, if you design databases even somewhat for a living, these tools will be woefully inadequate for your needs.

■ **Note** This chapter mainly covers the mechanics and language 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.

■ **Note** There are many types of models or diagrams: process models, data flow diagrams, data models, sequence diagrams, and others. For our 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 15 years has used it for that amount of time too.)

IDEF1X is based on Federal Information Processing Standards (FIPS) Publication 184, published September 21, 1993. To be fair, the other major mainstream 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:

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

■ **Note** At the time of this writing, the full specification for IDEF1X is available at `www.idef.com/idef1x-data-modeling-method/`. The exact URL of this specification is subject to change, and did since the previous edition of the book.

WOW! eBook
www.wowebook.org

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/data-modeler`
- *Toad Data Modeler*: `http://software.dell.com/products/toad-data-modeler/`
- *ER/Studio*: `www.embarcadero.com/products/er-studio`
- *Visible Analyst DB Engineer*: `www.visible.com/Products/Analyst/vadbengineer.htm`
- *Visio Enterprise Edition*: `www.microsoft.com/office/visio`

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

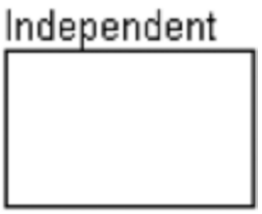
Entities

In the IDEF1X standard, entities 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, typically 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*. 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." Another term that is commonly used for the same purposes is "strong" for identifying, and "weak" for nonidentifying.

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 probably some sequence attribute like `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, 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 shown here:



WOW! eBook
www.wowebook.org

The dependent entity is drawn with rounded corners, as follows:

■ **Note** The concept of dependent and independent entities leads us to a bit of a chicken and egg paradox (not to mention, a fork in the road, also known as the don't write just before lunch principle). 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 and model 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 a perfect, easy-to-type 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.

Naming database objects is no different; actually, naming database objects clearly is more important than naming other programming objects, as your often *very* nontechnical end users will almost certainly get used to these names: the names given to entities will be used as documentation and 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. A central concern is whether to use plural or singular entity/table names. Both have merit, but one style has to be chosen and followed. 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, which also makes sense from another reasonable standpoint. It truly matters how you use the names. Plural names tend to need to be constantly made singular. Say you have a table that represents donuts. To discuss it, you would need to say awkward-sounding things such as "I have a donuts table. There are N donuts rows. One donuts row is related to one or more donuts_eaters rows." Singular names are appended to descriptions easily. "I have a donut table. I have N donut rows. One donut row is related to one or more donut_eater rows." By applying the pattern of adding the name to a scope, we can build up lots of documentation easily.

Each method has benefits and strong proponents, and plural or singular naming might be worth a few long discussions with fellow architects, but, honestly, 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

WOW! eBook
www.wowebook.org

go along. Any naming standard is better than no standard 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.
- *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.

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 most frequently 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.

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 interpreted 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 `tbl_Customer`. Prefixes like this are generally considered a bad practice for entities (and tables), because names in relational databases are almost always used in an obvious context. Using Hungarian notation is often 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 operator 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 list all of the different object types in the catalog (your results may vary; this query was executed against the `WideWorldImporters` database we will use for some of the examples in this book):

```
SELECT  DISTINCT type_desc
FROM    sys.objects
ORDER   BY type_desc;
```


WOW! eBook
www.wowebook.org

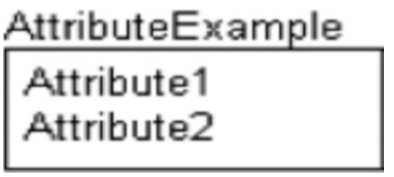
Here's the result:

```
type_desc
-----------------------------------------
CHECK_CONSTRAINT
DEFAULT_CONSTRAINT
FOREIGN_KEY_CONSTRAINT
INTERNAL_TABLE
PRIMARY_KEY_CONSTRAINT
SECURITY_POLICY
SEQUENCE_OBJECT
SERVICE_QUEUE
SQL_INLINE_TABLE_VALUED_FUNCTION
SQL_SCALAR_FUNCTION
SQL_STORED_PROCEDURE
SYSTEM_TABLE
TYPE_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 for a table and for IDEF1X). I'll cover the notation for keys in the following section.

WOW! eBook
www.wowebook.org

At this point, you would simply enter all of the attributes that you discover from the requirements (the next chapter will demonstrate this process). As I will demonstrate in Chapter 5, the attributes will be transformed a great deal during the normalization process, but the process is iterative, and the goal will be to capture the details that are discovered. 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 different entities depending on your actual system's needs.

■ **Note** Attribute naming is one place where I tend to deviate slightly from IDEF1X. The standard is that names are unique within a model, not just within a table. This tends to produce names that include the table name (or worse yet some table name prefix) followed by the attribute name, which can result in unwieldy, long names that look archaic.

Just as with entity names, there is no need to include Hungarian notation prefixes or suffixes in names to let you know it is a column. However, there is value in structuring names in a very straightforward way to let the reader know what the column means. The implementation details of the attribute can be retrieved from the system catalog if there is any question about it.

The format I use is loosely based on the concepts in ISO 11179, though there really is not much freely available about this standard available in a format that is worth referencing. Generally, the idea is that names include standard parts that are put together to form a standard-looking name. I will use the following parts in my names:

- *RoleName*: Optionally explains a specific role the attribute plays.
- *Attribute*: The primary purpose of the attribute being named. Optionally can be omitted, meaning it refers directly to the entity.
- *Classword*: A required term that identifies the primary usage of the column, in non-implementation-specific terms. Examples: `Name`, `Code`, `Amount`
- *Scale*: Optional to tell the user what the scale of the data is, like minutes, seconds, dollars, euros, etc.

Here are some examples of attribute names using some or all of the standard parts:

- `Name`: Simply uses a classword to denote a textual string that names the row value, but whether or not it is a varchar(30) or nvarchar(128) is immaterial (Without a role name, the name should apply directly to an instance of the entity. Example: `Company.Name` is the name of the company itself.)

WOW! eBook
www.wowebook.org

- `UserName`: An attribute and a classword, where the attribute tells the more specific use of the `Name` classword to indicate what type of name. (Example: `Company.UserName` would be the username the company uses. More context would be acquired from the name and purpose of the database.)
- `AdminstratorUserName`: A rolename added to the `UserName` attribute, identifying the specific role the user plays that is being named.
- `PledgeAmount`: Here the attribute `Pledge` is coupled with the money class, which is an amount of money, no matter the datatype which is used.
- `PledgeAmountEuros`: Indicates that for the `PledgeAmount`, this is an amount of money pledged, but with an atypical scale for the context of the database.
- `FirstPledgeAmountEuros`: Plays the role of the first `PledgeAmount` recorded in euros, unlike the previous which could be any pledge in order.
- `StockTickerCode`: Couples the `Code` classword (a short textual string) with the attribute part: `StockTicker`. So this is a short textual string representing a stock ticker. Note that each part of the name need not be a single word.
- `EndDate`: The `Date` classword says that this will not include a time part, and the attribute is the ending date for the row.
- `SaveTime`: Much like the previous attribute, only now it is a time, which will be treated as a point in time.

Almost all of the names used throughout the book will be in this format, unless I am explicitly attempting to demonstrate alternative naming. One of the most important parts of your database design is going to be consistent naming to assist users in understanding what you have built.

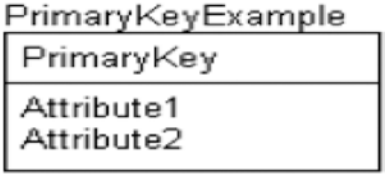
Next, we will go over the following aspects of attributes on your data model:

- 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).

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

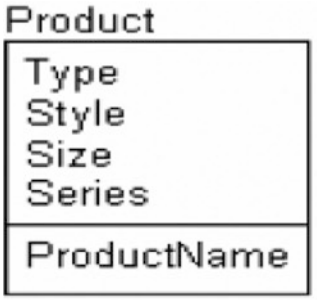


WOW! eBook
www.wowebook.org

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). 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 have the same `EmployeeNumber` value, thus requiring `EmployeeType` as part of the key. That's not a good practice perhaps, but no matter how much I try to describe perfect databases in this book, the business world is full of weird, archaic practices that you will have to incorporate into your models.) Having to go back repeatedly and change the attribute used for the primary key in the logical model 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.

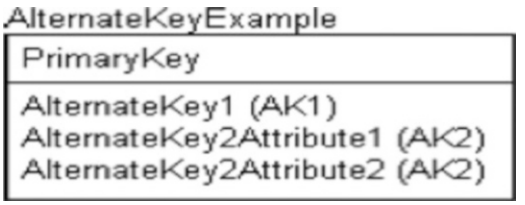
WOW! eBook
www.wowebook.org

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 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 philosophical debate at a table of data modelers—you have been warned, so start the debate after the dessert course).

Alternate Keys

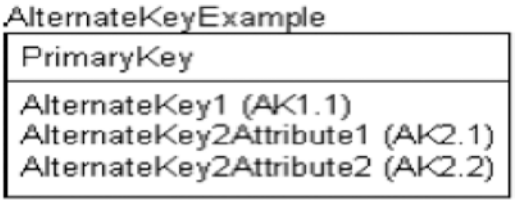
As defined in Chapter 1, an alternate key is a grouping of one or more attributes whose uniqueness needs to be guaranteed over all of the instances of the entity. Alternate keys do not have specific locations in the entity graphic like primary keys, nor are they typically migrated for any relationship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



WOW! eBook
www.wowebook.org

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:

One extension that the ERwin data modeling tool adds to this notation is shown here:



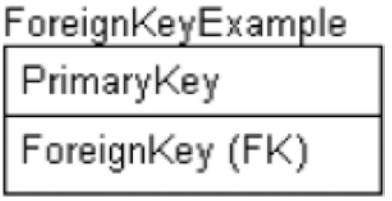
A position number notation is tacked onto the name of each key (`AK1` and `AK2`) to denote the position of the attribute in the key. In the logical model, technically, the order of attributes in the key should not be considered even if the tool does display them (unique is unique, regardless of key column 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 in the database, the order of columns *will almost certainly* become interesting for performance reasons, but uniqueness will be served no matter what the order of the columns of the key is.

■ **Note** Primary and unique constraints are implemented with indexes in SQL Server, but 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 design phases. Defer most performance tuning issues until you are coding and have enough data to really see what indexes are needed.

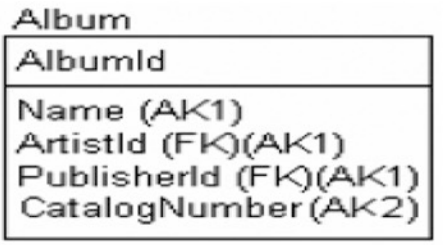
WOW! eBook
www.wowebook.org

Foreign Keys

Foreign key attributes, as mentioned, 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.

One tricky thing about foreign keys 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 (though some tools have this option available) 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 best practice at all, but it is possible and can make for interesting situations.

As you can see, 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.

WOW! eBook
www.wowebook.org

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)`
- `TextualFlag`: 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 a requirement 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. For example, a person's first name might be a domain. This is cool because you don't have to answer more than once questions such as "Hmm, how long to make a person's name?" or "What is the format of our part numbers?" After you make a decision, you just use what you have used before.

If it sounds unreasonable that you might argue about a datatype length, it is likely that you haven't been employed at a job without a nametag just yet. Programmers argue all of the time, but if you establish a standard after the first argument, you only have to have that argument once. Note too that most everything you want to store data on has been done before, so look to standards documents on the Web. For example, how long might a dialable phone number column be? The answer is 15 characters based on ITU-T E.164 (`http://searchnetworking.techtarget.com/definition/E164`). So making it 100 characters "just in case one day it is 6.66 times longer" is silly, not to mention bad for data integrity.

■ **Note** Defining common domains during design fights against another major terrible practice, the `string(200)` syndrome (or similar string length), 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. It is also the first step in producing databases with data integrity.

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.

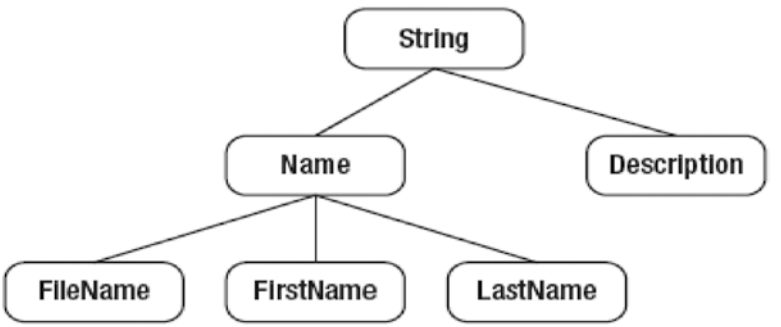
WOW! eBook
www.wowebook.org

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 database 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 physical 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 makes up a lot 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 your 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 more suitable datatype for a situation in the future, you can simply—at least from the model's standpoint—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.

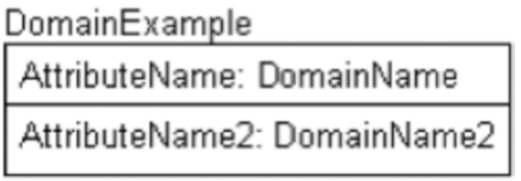
As an example of a domain hierarchy, consider this set of character string domains:



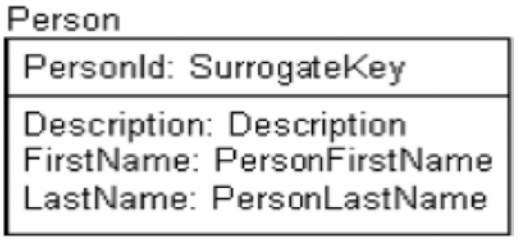
Here, `String` is the base domain from which you can then inherit `Name` and `Description`. `FileName`, `FirstName`, and `LastName` are inherited from `Name`. During logical modeling, this might seem like a lot of work for nothing, because most of these domains will share only a few basic details, such 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 finally make that blanket change from ANSI character sets to UNICODE, or to implement encryption on all personal notes–type attributes but not name ones.

WOW! eBook
www.wowebook.org

During logical modeling, domains might optionally be shown to the right of the attribute name in the entity (which is where you will eventually see the SQL Server datatype as well):



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:

1. `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.)
2. `Description`: Holds the description of "something" (can be 60 characters maximum).
3. `PersonFirstName`: A person's first name (30 characters maximum).
4. `PersonLastName`: A person's last name (50 characters maximum).

The choice of the length of name is an interesting one. I searched on Bing 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 that 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 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.

If you follow the naming standard I discussed earlier made up of RoleName + Attribute + Classword + Scale, you may find that your domains often share a lot of similarity with that hierarchy. `Name` is a classword, and `Name` is in our domain list. `FirstName` is an attribute and classword that also ends up as a domain. Domains, however, will often include things like size. `Name30Characters`, `Name60Characters`, etc. may not be good classwords, but they can be perfectly acceptable domains in some cases where you just want a class of names that are 30 characters, and don't want to make one domain per attribute. Domains are there for your sake, to make things easier for you, so do what makes more sense.

WOW! eBook
www.wowebook.org

Relationships

Up to this point, the visual constructs we have looked at have been pretty much the same across most data modeling methodologies. Entities are almost 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.

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 relationship concepts 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 nonidentifying*, when the nonidentifying relationship does not require a parent 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.

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 a database design diagram and not a completely simple one. A lot of information is related using a few dots and lines. Often it will help to look at the metadata that is represented in the graphical display to make sure it is clear (particularly if looking at a foreign modeling language!).

WOW! eBook
www.wowebook.org

■ **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.

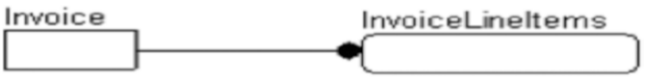
Identifying Relationships

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) is migrated to the primary key of the child. Hence, 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:



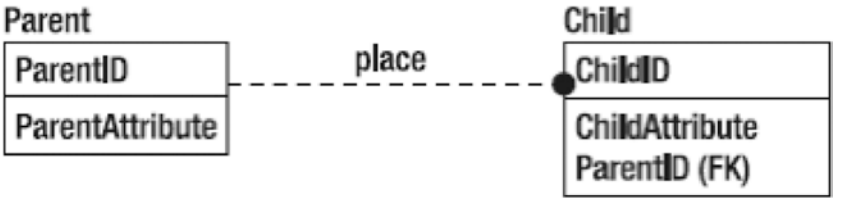
It can also be said that the line items are part of the parent, much like in an object you might have a property that is an array, but since all values are stored as scalars in SQL, we end up with more tables instead of different datatypes.

WOW! eBook
www.wowebook.org

Nonidentifying Relationships

The *nonidentifying relationship* indicates that the parent represents a more informational attribute of the child. When implementing the nonidentifying relationship, the primary key attribute is migrated as a non-primary key attribute 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`:

Now you can see that the attribute `ParentID` is migrated to the non-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.

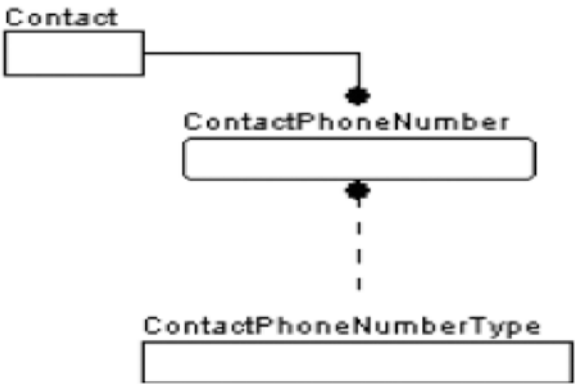
The difference between identifying and nonidentifying relationships can sometimes be tricky but is essential to understanding the relationship between tables and their keys. If the parent entity defines the need for the existence of the child (as stated in the previous section), then use an identifying 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. The `Contact` owns defines the phone number, and without the contact, there would be no need for the `ContactPhoneNumber` instance.

WOW! eBook
www.wowebook.org

- *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 would still be interesting, as the number still may have informational merit. However, a row associating a contact with a phone number would be useless information without the contact's existence.



The `ContactPhoneNumberType` entity is commonly known as a *domain entity* or *domain table*, as it serves to implement an attribute's 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 could be mandatory. The difference in the implemented table will be whether or not the child table's foreign key columns will allow `NULL`s. 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*.

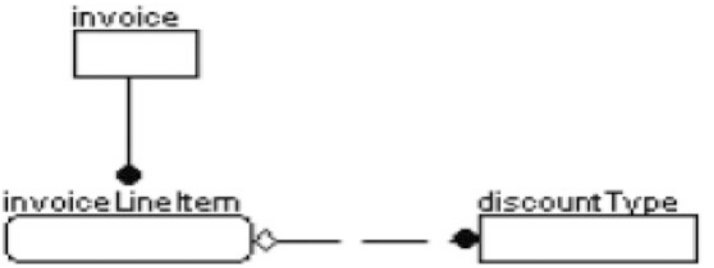
The optional case is signified by an open diamond at the opposite end of the dashed line from the black circle, as shown here:

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 or greater, but a mandatory relationship must have a cardinality of one or greater (as defined in Chapter 1, *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 as well).

WOW! eBook
www.wowebook.org

■ **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 SQL Server as well.

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" or "unknown." An example of such a mandatory relationship could be genre to movie in a movie rental system database:

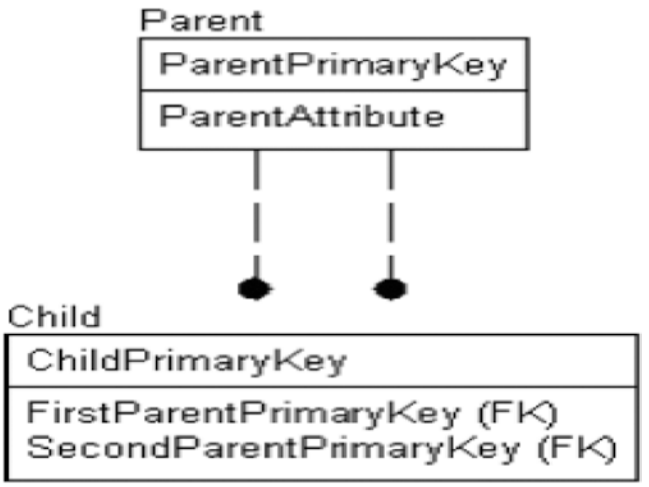
The relationship is `genre <classifies> movie`, where the `genre` entity represents the "one" and `movie` represents the "many" in the one-to-many relationship. Every movie being rented must have a genre, so that it can be organized in the inventory and then placed on the appropriate rental shelf. If the movie is new, but the genre wasn't yet known, a row in the genre object with a genre of "New" or "Unknown" could be used. Whether or not to use an optional relationship, or to manufacture a row that means the lack of a value, is discussed quite frequently. Generally, it is frowned upon in nonreporting databases to come up with a value that means "There is not a real related value" because it can cause confusion. Users who are writing a query and end up seeing sales to a "Missing Customer" might not be sure if that is the name of a funky new restaurant or a manufactured row. In reporting you generally are just looking at groupings, so it is less of an issue and you will design some way to make the row sort to the top or the bottom of the list.

Role Names

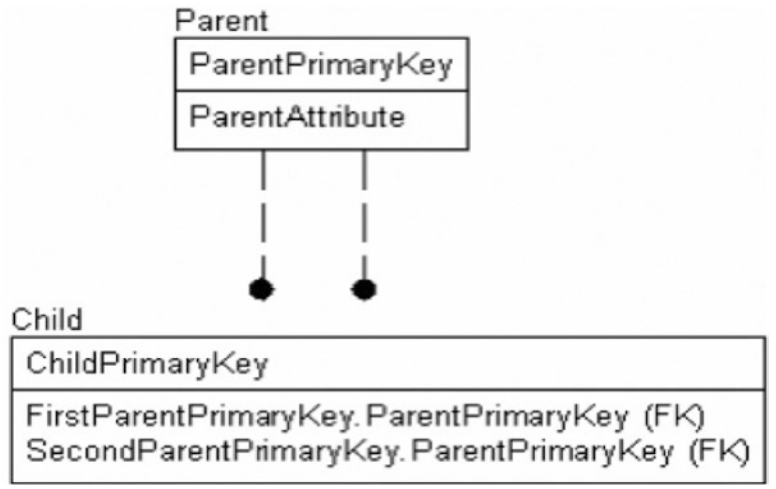
A *role name* is an alternative name you can give an attribute when it is used as a foreign key. The purpose of a role name is to clarify the usage of a migrated key, because either the parent entity is generic and a more

WOW! eBook
www.wowebook.org

specific name is needed or the same entity has multiple relationships to the same entity. As attribute names must be unique, assigning different names for the child foreign key references is often necessary. Consider these tables:

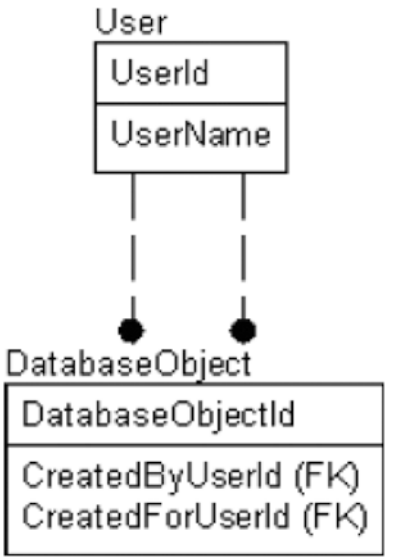


In this diagram, the `Parent` and `Child` entities share two relationships, and the migrated attributes have been role named as `FirstParentPrimaryKey` and `SecondParentPrimaryKey`. In diagrams, you can indicate the original name of the migrated attribute after the role name, separated by a period (`.`),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



WOW! eBook
www.wowebook.org

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 key's 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 is the number of child instances that can be inserted for each parent instance of that relationship. A lot of people slough off this topic in requirements and design because it can be difficult to implement. However, our initial goal is to represent the requirements in our data models, and leave discussion of data constraint implementation to later (we will start discussing implementation in Chapter 6 when we implement our first database, and discuss it even more in Chapter 7, the data protection chapter, and throughout the latter half of the book).

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*

WOW! eBook
www.wowebook.org

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)*

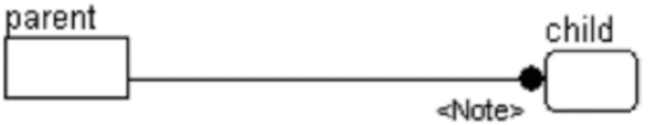
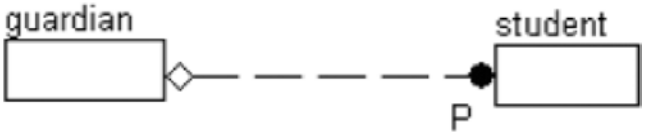


Figure 3-6. *Specialized note describing the cardinality*

For example, a possible use for the one-to-one or more (see Figure 3-2) might be to represent the relationship between a guardian and a student in an elementary school:



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.

WOW! eBook
www.wowebook.org

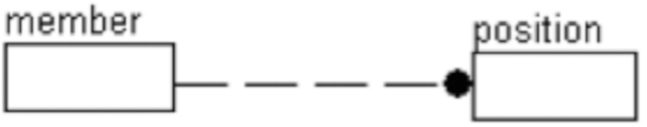


Figure 3-7. *One-to-many allows unlimited positions for the member*

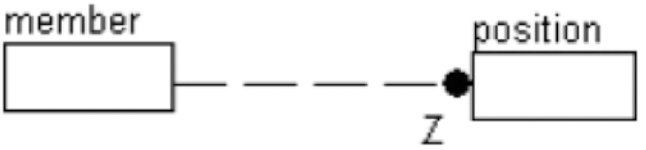


Figure 3-8. *One-to-one allows only one position per member*

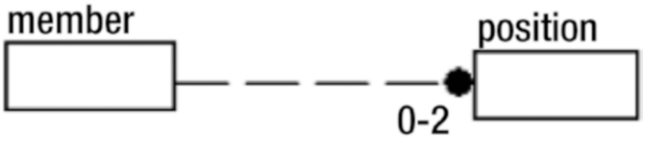


Figure 3-9. *A one-to-zero, one-to-one, or one-to-two relationship specifies a limit of two positions per member*

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 very similar, but the Z or 0–2 is important in signifying the cardinality.

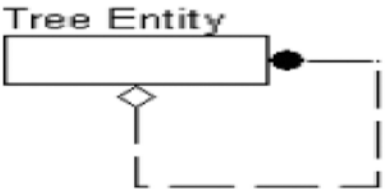
■ **Note** It is not an overly common occurrence that I have needed anything other than the basic one-to-many, one-to-zero-or-one relationship types, but your experience may lend itself to the specialized relationship cardinalities.

Recursive Relationships

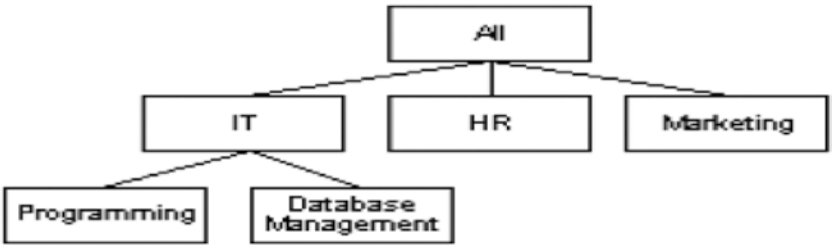
One of the more difficult—and often important—relationship types to implement is the *recursive relationship*, also known as a *self-join*, *hierarchical*, *self-referencing*, or *self-relationship* (I have even heard them referred to as *fish-hook relationships*, but that name always seems silly to me). It is used to model some form of hierarchy, where one row is related to one and only one parent. The "recursive" part of the name references the method of traversing the structure. In Chapter 8 we will look in more depth at hierarchies, some that use a recursive relationship, some that do not.

The recursive relationship is modeled by drawing a nonidentifying 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" or "referenced" to the front of the attribute name is useful if no natural naming is available.)

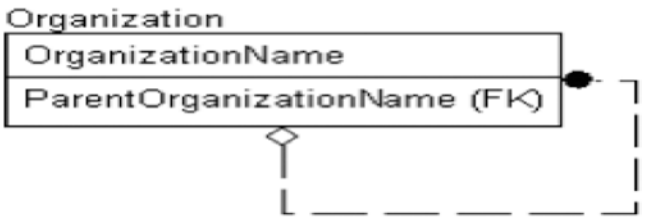
WOW! eBook
www.wowebook.org



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:



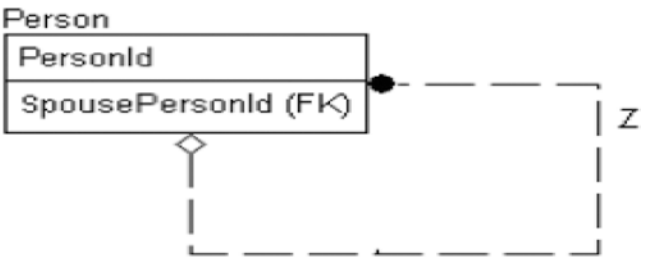
Here is the sample data for this table:

```
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`.

WOW! eBook
www.wowebook.org

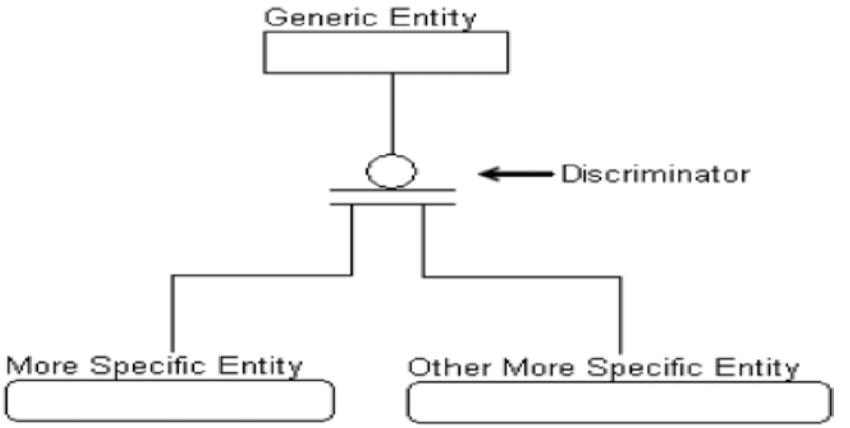
As a final example, consider the case of a `Person` entity. If you wanted to associate a person with a single other person as the first person's current spouse, you might design the following:



Notice that this is a one-to-zero-or-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 that I said "current" spouse earlier in the section. If you need to know history of changes in hierarchy, you will need some of the more complex versions of modeling hierarchies that we will cover in some detail in Chapter 8 when we discuss various modeling patterns and techniques. In this chapter, it is strictly important that you can grasp what the hierarchical relationship looks like on a data model.

Subtypes

Subtypes (also referred to as *categorization relationships*) are a special type of one-to-zero-or-one relationship used to indicate whether one entity is a specific type of a generic entity. It is similar to the concept of inheritance in object-oriented programming (if a lot clunkier to work with). Note in the following diagram 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.

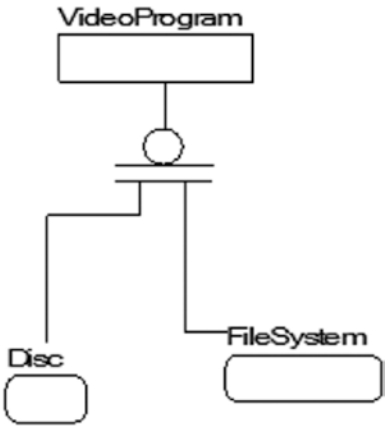


WOW! eBook
www.wowebook.org

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 `Disc` 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`).

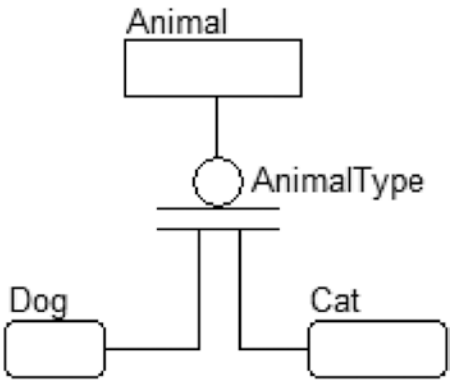
There are two distinct category types: complete and incomplete. The *complete* set of categories is modeled with a double line on the discriminator, and the *incomplete* set is modeled with a single line (see Figure 3-10).

WOW! eBook
www.wowebook.org

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.

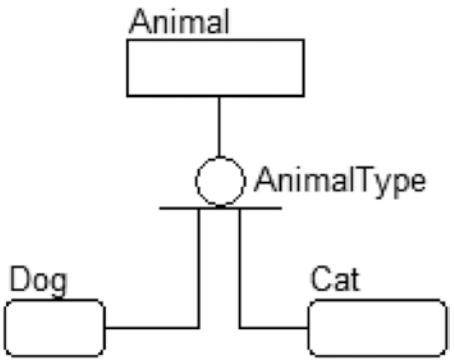
For example, you might have a complete set of categories like this:



This relationship indicates that "An `Animal` *must* be either a `Dog` or a `Cat`." (Clearly true only in the context of the requirements for this company, naturally. However, what it doesn't show us is first, do we have to know that the animal type is known? There may be no rows in `Dog` or `Cat` at all, and animal type may be `NULL`.

WOW! eBook
www.wowebook.org

However, what if this were modeled as an incomplete set of categories:



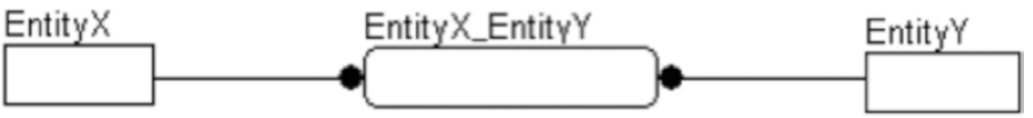
Now in our database, we may have `Dog`, `Cat`, `Fish`, `Lizard`, etc., but we only have specific information recorded for `Dog` and `Cat`.

Additionally, a concept that is mentioned occasionally is whether a subclass is *exclusive* or not exclusive. For the animal, `Animal` can clearly only be a `Dog`, `Cat`, etc., not both. But consider the common subclass of `Person` to `Customer`, `Employee`, and `Manager`. A person can be a manager, who is also an employee, and it may be advantageous to track their customer relationship in the same database from the subclass structure. This would not be an exclusive subtype.

Many-to-Many Relationships

The many-to-many relationship is also known as the *nonspecific relationship*, which is actually a better name, but far less well known. Having lots of many-to-many relationships in the data model is common, though they will be modeled as something different when you are building your physical model. These relationships are modeled by a line with a solid black dot on both ends:

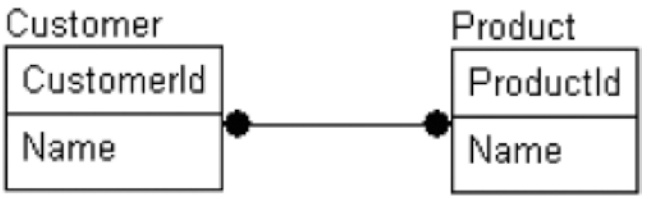
There is one real problem with modeling a many-to-many relationship, even in the logical model: it is often necessary to have more information about the relationship than that simply many `EntityX` instances are connected to many `EntityY` instances. So, much like any many-to-many relationship will be implemented, the relationship is usually modeled as follows very soon after its discovery:



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

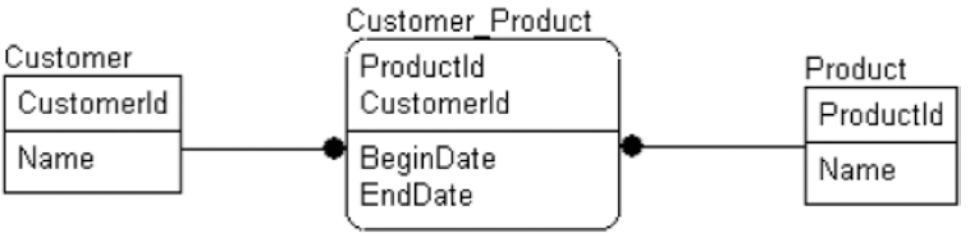
WOW! eBook
www.wowebook.org

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 because in most cases, companies don't create specific products for specific customers; rather, any customer can purchase any of the company's products, or certainly a subset of products that are also orderable by other customers. At this point in the modeling, it is likely 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, making for a more interesting modeling requirement.

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 in anything other than a simple conceptual model, so you will need to look for entities modeled like this to be able to discern them.

■ **Note** You can't implement a many-to-many relationship in SQL without using a table for the resolution because there is no way to migrate keys both ways. In the database, you are required to implement all many-to-many relationships using an associative table.

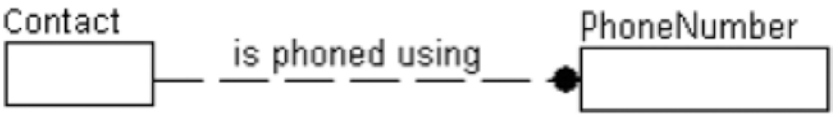
Verb Phrases (Relationship Names)

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:

WOW! eBook
www.wowebook.org

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.

For example, consider the following relationship:

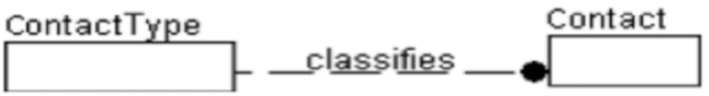


It would be read as "one `Contact` is phoned using zero, one, or more `PhoneNumbers`."

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," or perhaps "If the contact has a phone number, the contact can be phoned with one or more phone numbers." You don't want to simply use "have" for all verb phrases, as the goal of the verb phrase is to capture the essence of how the data will be used and provide documentation that doesn't need editing to be semantically understandable (even if not tremendously impressive prose).

Being able to read the relationship helps you to notice obvious problems. For instance, consider the following relationship:

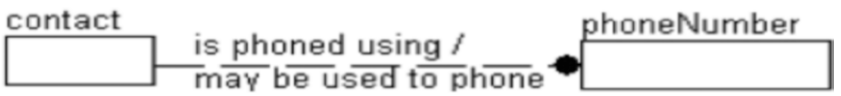
It looks fine at first glance, but when read as "one `ContactType` classifies zero or one `Contacts`," it doesn't make logical sense (since `Contact` requires a `ContactType`, you would need the same number or more rows in a `ContactType` table). This would be properly modeled as follows:



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:



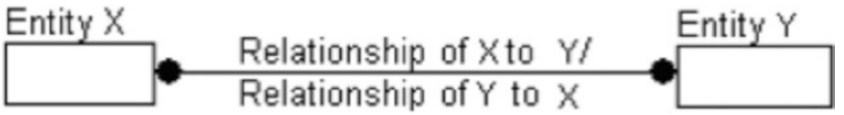
WOW! eBook
www.wowebook.org

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.

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 read the verb phrase written below the line from right to left.



■ **Note** Taking the time to define verb phrases can be a hard sell, 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 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. So far we have only really discussed things you can see in the graphical part of the model. You can get a great start on understanding the database from the picture, 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 as metadata in a modeling tool (preferably) or in some sort of document that will be easy to maintain in the future.

WOW! eBook
www.wowebook.org

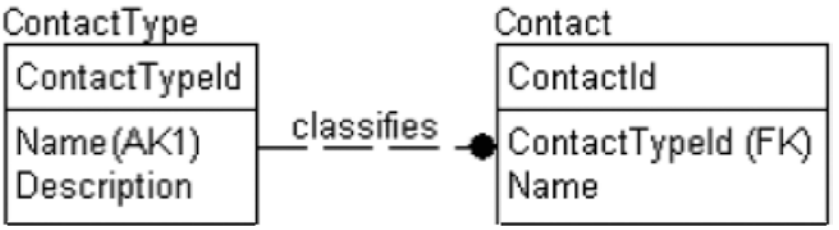
You should start creating this descriptive text by asking questions such as the following:

- 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?

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. It is also a good idea to avoid a lot of examples, as they may change, while the model itself may not need to change.

Maintaining good descriptive information is roughly 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. For most people, being able to go back and review notes that were taken about each object and why things were implemented is invaluable, which is especially true for organizations that hire 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 stored 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

WOW! eBook
www.wowebook.org

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 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 may 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.)

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

---

■ **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 (www.agiledata.org/essays/umlDataModelingProfile.html), and on IBM's Rational UML documentation site (www.ibm.com/software/rational), among many others. (The typical caveat that these URLs are apt to change applies.) SQL Server Management Studio includes some rudimentary modeling capabilities that can show you the model of a physical database as well.

Information Engineering

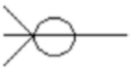
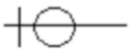
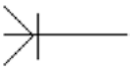
The *information engineering (IE)* methodology is well known and widely used (it would be a tossup as to which methodology is most common, IE or IDEF1X). Like IDEF1X, it does a very good job of displaying the necessary information in a clean, compact manner that is easy to follow. The biggest difference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

WOW! eBook
www.wowebook.org

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 often 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 typically 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.

Table 3-3. *Information Engineering Symbols*

Symbol	Relationship Type
	Zero, One, Or Many
	Zero Or One
	At least one or many
	One and Only One

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

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

WOW! eBook
www.wowebook.org

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*

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 circles, 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 very different from IDEF1X, but it's pretty easy to follow and largely self-explanatory. You will seldom see this methodology corporately, as it is mainly used in the academic world, but since quite a few of these types of diagrams are on the Internet, it's good to understand the basics of the methodology. In Figure 3-15 is a very simple Chen ERD diagram showing the basic constructs.

WOW! eBook
www.wowebook.org

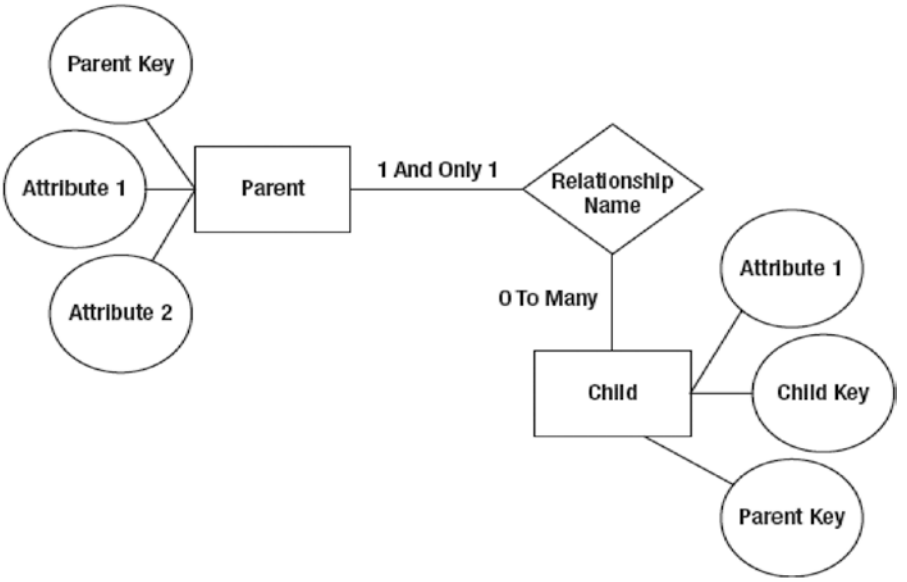


Figure 3-15. *Example Chen ERD diagram*

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.

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 an Internet 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.

WOW! eBook
www.wowebook.org

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 correctly. 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 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*: While it is perfectly acceptable as a practice, it is 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.

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 a 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:

WOW! eBook
www.wowebook.org

- 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 3-16.

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.

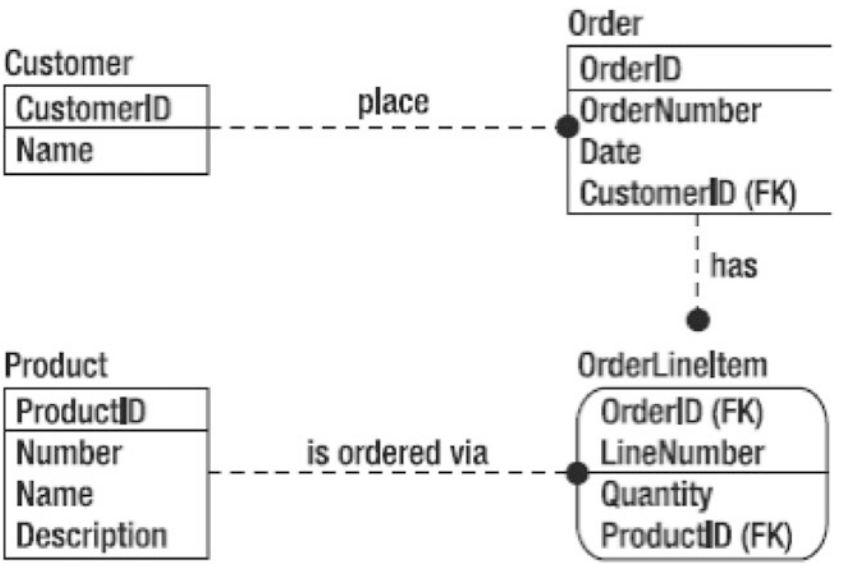


Figure 3-16. *Reading this basic model is not difficult at all, if you simply apply the explanations from this chapter*

WOW! eBook
www.wowebook.org

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 to 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.

WOW! eBook
www.wowebook.org

CHAPTER 4

Conceptual and Logical Data Model Production

I have a new philosophy. I'm only going to dread one day at a time.

—Charles M. Schulz,
cartoonist best known for the comic strip Peanuts

In this chapter, we are going to really wind things up and begin to apply the skills that were covered in the previous chapters and begin creating a data model. It won't likely be identical to 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. Personally, I both love and loathe this particular step in the process because this is where things get complicated. All of the requirements and documents need to be considered. The architects and programmers from all disciplines, ideally, will be collaborating to achieve a data model and designs for the user and back-end experiences. This step is where you will also get to be the most artistic, looking for unique and interesting solutions to a variety of data needs.

Ideally, 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 and verbally. In either case, the fun part starts now: sifting through all theses 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. It is very difficult to get requirements perfect from any human being. In this chapter, I am going to assume the requirements are perfect for simplicity's sake, but expect things to shift to meet requirements that were never captured.

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_4

WOW! eBook
www.wowebook.org

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 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*: Looking for natural relationships between high-level entities. Relationships between entities are what make entities useful.
- *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 and test that you can meet the requirements, but not so much that a great investment has been made.

Once everyone feels like the entities and relationships make sense, 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. By the time you have completed this task, you should have a strong sense that the user has what they need.

This 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 model that is ready to be finalized as 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 presented 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 may 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 through a complex math problem, in that you are solving a big problem 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 do 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 a physical model. However, the more you know about how a proper database should look, 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.

WOW! eBook
www.wowebook.org

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 among hundreds. (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.)

> *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 and where 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 technology professional 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 simplified. 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, I will present smaller examples to demonstrate independent concepts in modeling that have been trimmed down to only the concepts needed.

Building the Conceptual Model

The conceptual model is all about the big picture of the model. What is being modeled? How should the resulting database look? Details like how data will be stored should be left alone for now. We want to know, "What are the concepts the customer wants to store data about?" Customers, Dentists, Insurance Policies, etc. and how they are related are the essential parts of this process. Customers have Insurance Policies, and other relationships give you the skeleton of the database.

WOW! eBook
www.wowebook.org

If you have ever been involved with building a building of some sort, there are many iterations in the process. You start by sketching out the basic structure that the customer wants, and refine it over and over until you have a solid understanding of what the customer wants. You may not know where the building will be built, probably don't know what the materials will be, and certainly have no idea where the plugs in the kitchen will be located. Everything is changeable, but usually once the design shows the concepts of the final building and the customer approves it, the foundation for success has been laid.

In the database design process, our goal will be to understand the types of data the customer needs to store, and how things are related to one another. When we complete the conceptual model, we will have the framework of our database completed, and will be ready to fill in the details.

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, doing so 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
- *Places*: Dental office, patient's home, hospital
- *Objects*: A dental tool, stickers for the kids, toothpaste
- *Ideas*: 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. 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 still a click and drag away.

Persons

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 your actual users required to create a user 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.

WOW! eBook
www.wowebook.org

In our example, three people entities can be found by reading through our example scenario—patients, dentists, and hygienists:

> *. . . 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). 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.

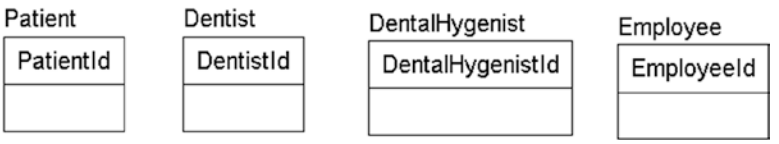


Figure 4-1. *Four entities that make up our initial model*

■ **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 regard 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 laypeople 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 expect that there's address information about the offices, and probably phone numbers, staffing concerns, and so on that the user may be interested in capturing information about. 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. "Expecting" is just slightly informed guessing, so verify all expectations with the client.

WOW! eBook
www.wowebook.org

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

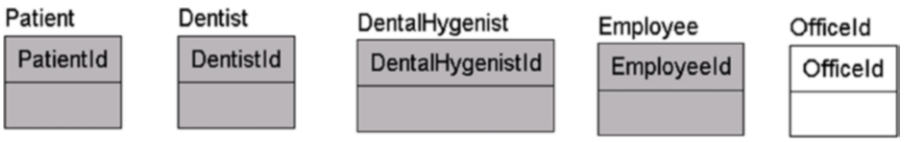


Figure 4-2. *Added Office as an entity*

■ **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 additions are unshaded.

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:

1. *Tube size*: Perhaps the length of the tube or the amount in grams
2. *Brand*: Colgate, Crest, or some other brand/lack of brand (which should not feel slighted by the lack of inclusion in my book)
3. *Format*: Metal or plastic tube, pump, and so on
4. *Flavor*: Mint, bubble gum (the nastiest of all flavors), cinnamon, and orange
5. *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 samples they have been bribed with from the dental companies and give them out to their patients (to make them feel better about the metal against enamel experience they have just gone through).

One of the first, extremely important 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 that makes it necessary, or that the client will change their processes to match your design. If you have good ideas, they might change, 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.

WOW! eBook
www.wowebook.org

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

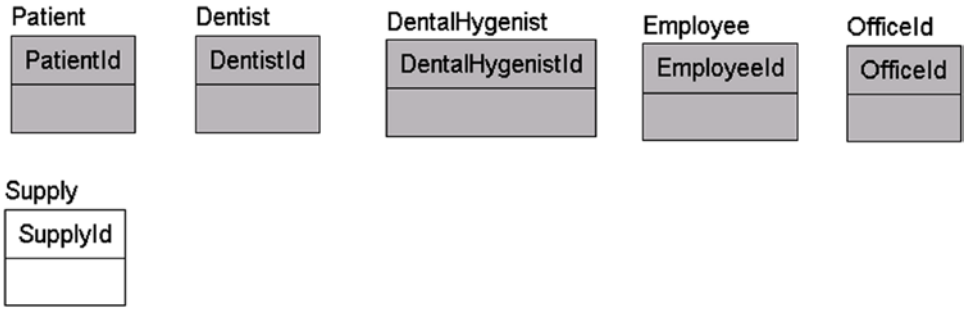


Figure 4-3. *Added the Supply entity*

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.

■ **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.

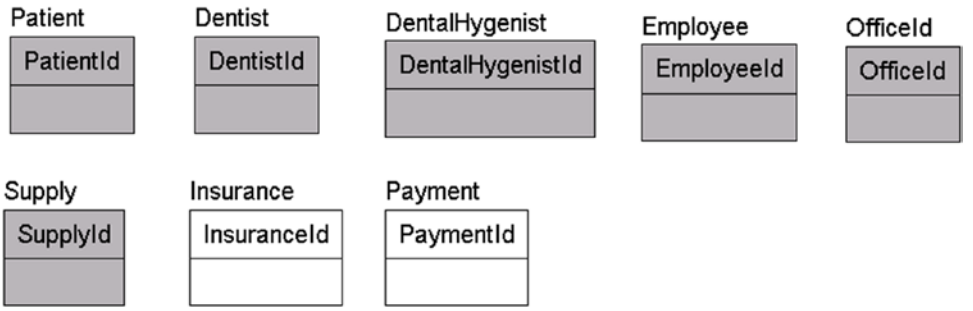


Figure 4-4. *Added the Insurance and Payment entities*

WOW! eBook
www.wowebook.org

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 Interwebs 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 to make cupcakes to take to a child's school (have some birthday celery, classmates!). 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 there is a specific business rule governing the method of or tracking the delivery (and even still things change faster than database implementations, so be wary of hard-coding business rules in a nonchangeable manner). At this point, just identify the entities (`Invoice` in this instance) and move along; again, it usually isn't worth it to spend too much time guessing how the data will be used.

Next up, we have the following:

> *. . . appointments, **alerting** the patients when and where 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, `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 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, e-mailed you, and texted you on June 14th to no avail using the number that you provided us!" It may also be important that you track the addresses that were used for alerts, in case the primary numbers change so you can know what address was actually alerted, but more on that later in the chapter.

■ **Note** If you are alert, you probably are thinking that `Appointment`, `Email`, and `Phone` are all entity possibilities, and you would be right. In my teaching process here, I am looking at the types one at a time to make a point. In the real process, you would just look for nouns linearly through the text and enhancing in a very set manner so this chapter isn't larger than my allocation of pages for the entire book.

WOW! eBook
www.wowebook.org

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

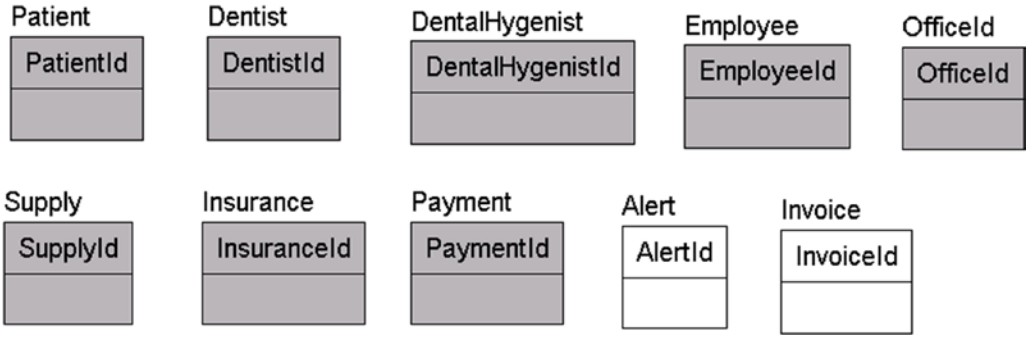


Figure 4-5. *Added the Alert and Invoice entities*

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 could be an attribute of the person, it's likely more than that. So, we add a `Family` entity, as shown in Figure 4-6. Remember that anything added to the conceptual model can be removed later.

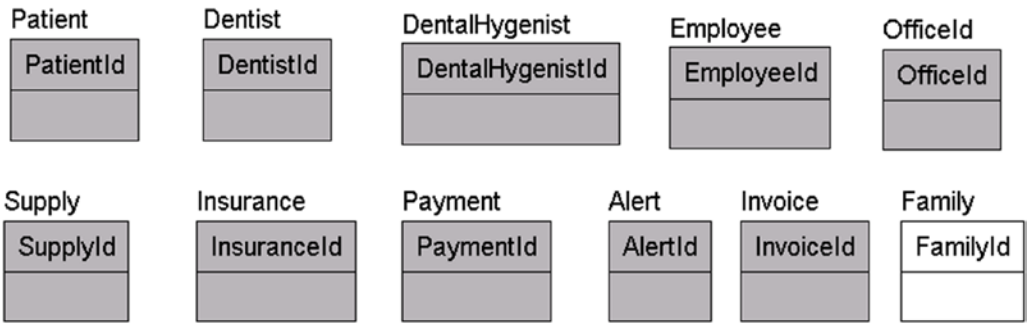


Figure 4-6. *Added the Family entity*

WOW! eBook
www.wowebook.org

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, 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 nondental 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.

WOW! eBook
www.wowebook.org

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 is one 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!" (Or perhaps even more importantly, given enough data from enough people, we can finally get a good idea of what course of drugs cures diseases, rather than simply using small-scale experiments. Keep in mind that the more data you can get into your database, the more likely that data can be used for data mining at some time in the future.) 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.

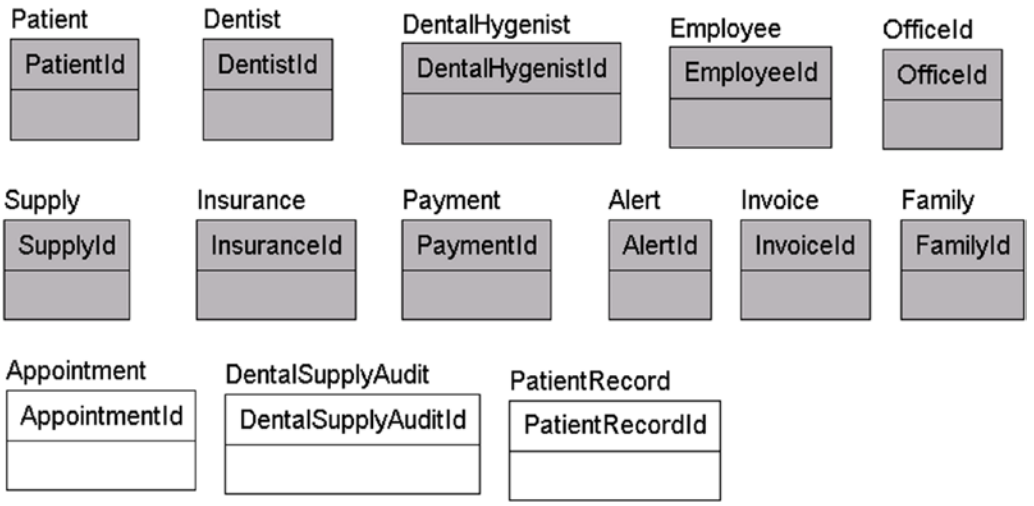


Figure 4-7. *Added the Appointment, DentalSupplyAudit, and PatientRecord entities*

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 generally best to just put it on the model as you have found it and do the refinement later when you feel you have everything needed to model it right.

WOW! eBook
www.wowebook.org

This is especially true when you are working from documents and not with the users directly, a practice that is becoming less prevalent, but not completely extinct.

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.

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.
`DentalHygienist`	People	People who do the routine 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 types of employees.
`Office`	Places	Locations where the dentists do their business. They have multiple offices to deal with and schedule patients for.
`Supply`	Objects	Supplies for the running of the dental operations of the office. Examples given were sample items, such as toothpaste or toothbrushes, plus there was mention of dental supplies as the supplies that the dentists and hygienists use to perform their jobs.
`Insurance`	Idea	Used by patients to pay for the dental services rendered.
`Payment`	Idea	Money received 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	A communication made to tell patient of an impending appointment.
`DentalSupplyAudit`	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.
`PatientRecord`	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.

WOW! eBook
www.wowebook.org

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 the set 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 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 when I did something similar for another client." Every client is unique and has its own way of doing stuff. The most important tools you will need to use are your eyes and ears.

Identifying 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. Just realize that 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 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:

Simple: One instance is related to one or more child instances. 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. In the data modeling chapter (Chapter 3), we referred to this type of relationship as a categorization relationship.

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

WOW! eBook
www.wowebook.org

Simple Relationships

In this section, I discuss some of the types of associations that you might uncover along the way as you are modeling relationships. Another common term for a simple relationships is often 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 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, which degrades the value of the verb phrase (sometimes, however, it really just is the best verb phrase).

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

- *Connection*: An association association between two things, like a person and their driver's license or car. This is the most generic of all relationships, and will generally cover any sort of ownership or association between entities.
- *Transaction*: more generically, this could be thought of as an interaction relationship. For example, a customer pays a bill or makes a 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.
- *Domain*: A type of relationship that you may possibly uncover in early modeling is a domain type. It is used to implement the domain for an attribute where more than a single attribute seems useful. I will not demonstrate this relationship type in the chapter, but we will look at domain relationships in later chapters (they are most commonly used as an implementation tool).

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

Connection

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.

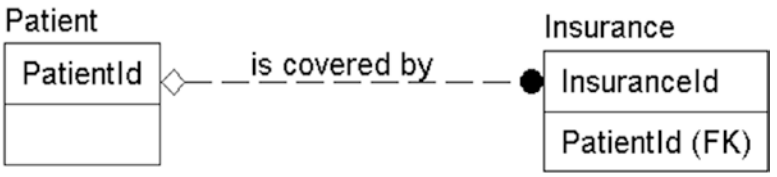


Figure 4-8. *Added the relationship between the Patient and Insurance entities*

WOW! eBook
www.wowebook.org

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.

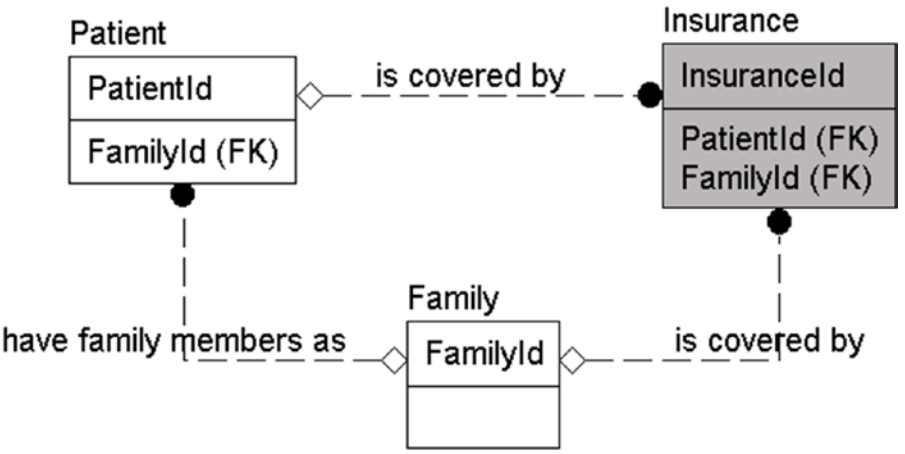


Figure 4-9. *Relationships added among the Patient, Insurance, and Family entities*

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

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

In this case, make note that each dental office will have appointments. Clearly, an appointment can be for only a single dental office, so this is not a many-to-many relationship. One of the attributes of an event type of entity is often 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. However, it is certain that appointments must be made at the office, so the relationship between `Office` and `Appointment` is required. Now add the relationship shown in Figure 4-10.

WOW! eBook
www.wowebook.org

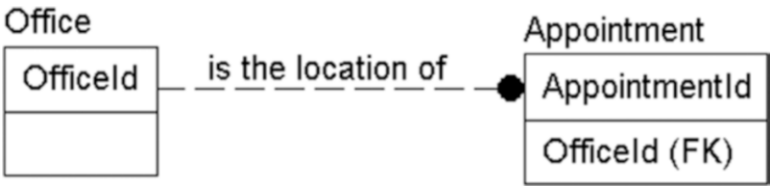


Figure 4-10. *Relationship added between the Office and Appointment entities*

Transactions

Transactions are probably the most common type of relationships in databases. Almost every database will have some way of recording interactions with another entity instance. For example, some very common transaction are simply customers making purchases, payments, and so on. 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.

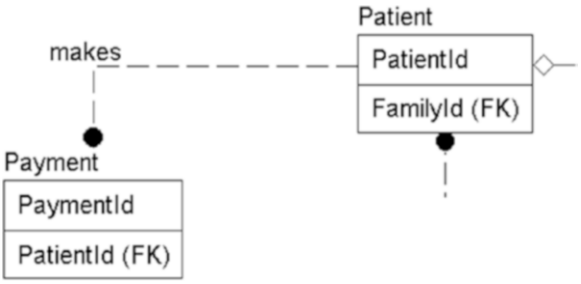


Figure 4-11. *Relationship added between the Patient and Appointment entities*

Multivalued Attributes and Domains

During the early phases of modeling, it is far less likely to discover multivalued attribute and domain 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.

For example, in terms of multivalued attributes: `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. For domain relationships, you can think of status values for most of the aforementioned entities. While a domain of "Active" and "Inactive" may not elevate to the need of a domain table, if we also want to associate a description with this value ("Active patients for more than 5 years get better goodies"), or perhaps include some process to take when the patient is in the status, having a table of possible status values would make that possible. In some cases, these types of complex domains may be shown on the conceptual model, but when a designer does so, it usually is because the designer is overcomplicating the process of conceptual modeling.

WOW! eBook
www.wowebook.org

As such, I won't come up with any examples of domain or multivalued attribute relationships in the example paragraphs, but we will cover this topic in more depth 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****, 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, but none listed at this point. Figure 4-12 represents this relationship, which is modeled using the subtype relationship type. Note that this is modeled as a complete subtype, meaning every `Employee` instance would either have a corresponding `Dentist` or `DentalHygenist` instance.

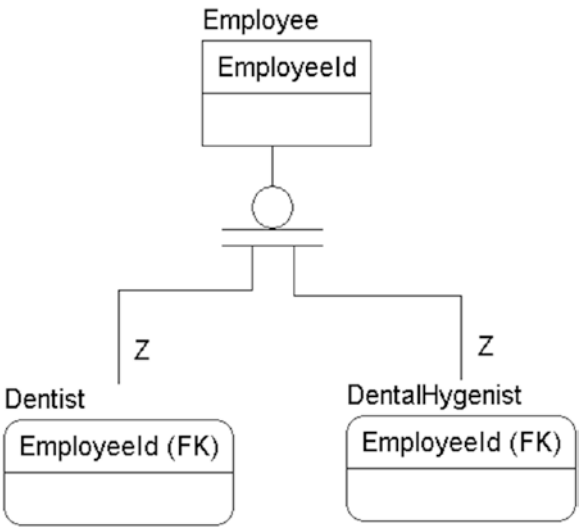


Figure 4-12. *Identified subtyped relationship between the Employee, Dentist, and DentalHygienist entities*

WOW! eBook
www.wowebook.org

■ **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.

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, each of the 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).

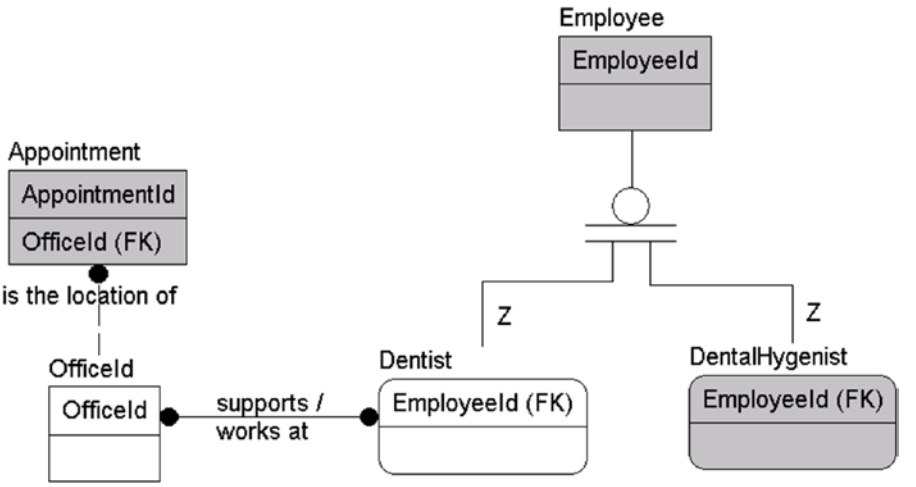


Figure 4-13. *Added a many-to-many relationship between Dentist and Office*

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 (and hopefully your requirements gatherer will have that discussion immediately. . . mine did not!). 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.

WOW! eBook
www.wowebook.org

There 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.

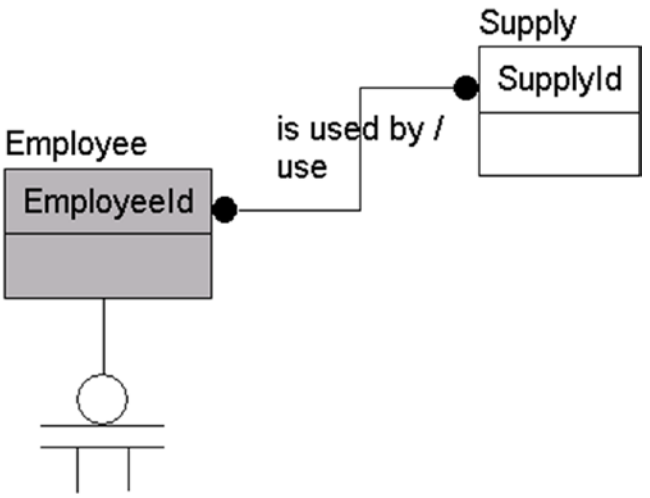


Figure 4-14. *Added a many-to-many relationship between the Supply and Employee entities*

I'm also going to remove the `DentalSupplyAudit` entity, because it's becoming clear that this entity is a report, and we will figure out how to solve this need later in the process. What we know from here is that employees use supplies, and we need to capture that that is happening.

WOW! eBook
www.wowebook.org

Listing Relationships

Figure 4-15 shows the model so far.

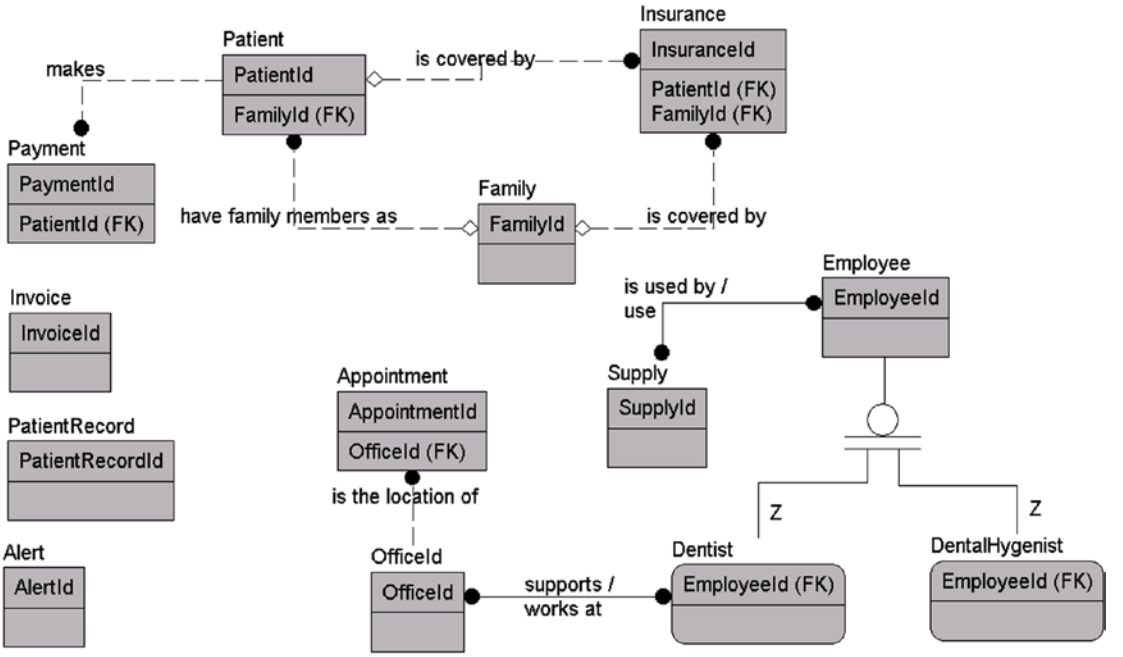


Figure 4-15. *The model so far*

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. *Initial 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.

(*continued*)

WOW! eBook
www.wowebook.org

Table 4-2. (*continued*)

Entity	Type	Description
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.
Dentist	People	People who do the most skilled work at the dental office. Several dentists work for the client's practice.
	Works at many Offices	Dentists can work at multiple offices.
	Is an Employee	Dentists have some of the attributes of all employees.
	Works during Appointments	Appointments might require the services of one dentist.
DentalHygienist	People	People who do the routine 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.
Supply	Objects	Supplies for the running of the dental operations of the office. Examples given were sample items, such as toothpaste or toothbrushes, plus there was mention of dental supplies as the supplies that the dentists and hygienists use to perform their jobs.
	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 received from insurance or patients (or both) to pay for services.

(*continued*)

WOW! eBook
www.wowebook.org

Table 4-2. (*continued*)

Entity	Type	Description
`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.

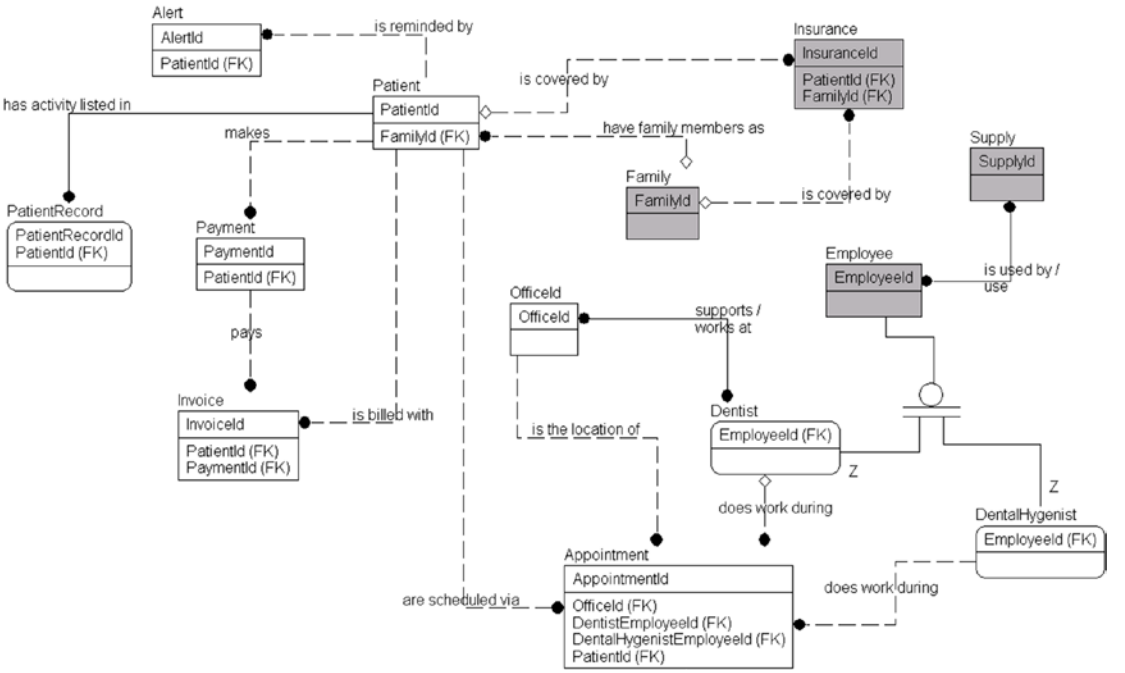


Figure 4-16. *The final conceptual model*

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)

WOW! eBook
www.wowebook.org

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.

Bear in mind that the only 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.

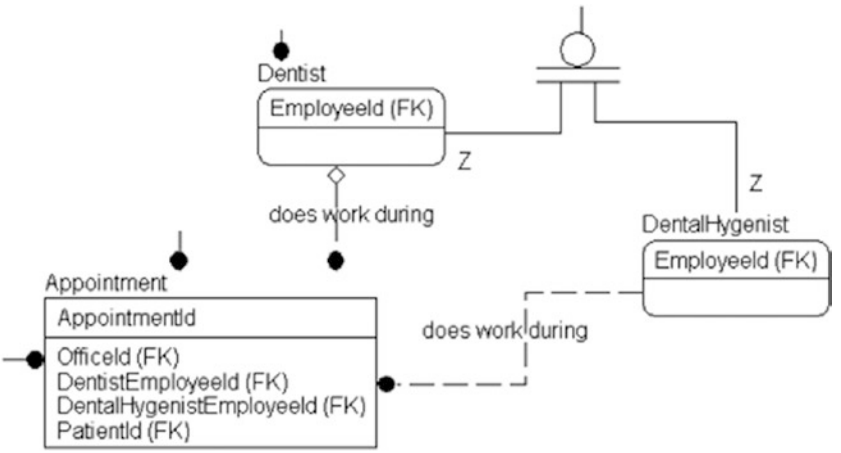


Figure 4-17. *Appointment entity with a role named EmployeeRelationship*

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.

The `Appointment` entity is the best example in my tiny example diagram of how the model shows you some basic makeup of the entity, as we can now see that for an appointment, we need an office, a patient, a hygienist, and sometimes a dentist available (since the relationship is optional). None of these things really defines an appointment, so it is still an independent entity, but those attributes are critical.

Testing the Conceptual Model

Before stamping "done" on the conceptual model, this is the time to take one last trip through the requirements and make sure that they can be met by the entities in your model. We will not spend too much time here in the text covering what this means, because it is basically the same process you have already gone through, iterating until you don't find anything that needs to be changed. Earlier in the chapter we took the following requirement:

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

Now we need to make sure that if something has changed with the entity we had created that it still meets the need. The test is likely as simple as a listing like:

1. Create a `Patient` instance.
2. Create a related `Insurance` instance to represent their insurance policy.

WOW! eBook
www.wowebook.org

Seeing that every requirement can be met at a high level will help you see if your model is prepared to meet all of the requirements that you have expected it to. It is possible that you may have changed an entity to something that will no longer meet the requirements, but you are still at a great place in the process where changes require zero coding.

Building the Logical Model

In this section, with the conceptual model completed and tested, the next steps in the modeling process are going to be considerably easier. Unlike the conceptual model, where you focus on the big picture, now we want to focus on the details. For the most part, the most difficult part of the data modeling process is past (getting a handle on the customer's needs). You will likely add more entities to the model, but usually the entities you will identify are really just to implement a complex (nonscalar) attribute. For example, if the attribute was phone numbers, you may need a new entity to allow multiple phone numbers.

Completing the house building analogy from the "Conceptual Model" section, when the logical model is completed, we will have fleshed out blueprints for the building. We will know the rooms, windows, and other features that the customer expects to have available. Things may change when the house is built, and similarly, when we erect a database we may need to tweak the design to make sure it will work in all of the scenarios we need. In Chapter 6 we will create a database and start testing in reality.

In this section we will continue the model we started in the main section of the chapter, adding attributes, identifying rules, processes, etc. to try to make sure we understand all of the data needs.

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 entities we're trying to represent, or—to put this into more computing-like terms—the properties of our 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/or 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 so much as identifying ways that a user may search for a particular instance.
- *Descriptive information*: Information used to describe something about the entity, such as color, status, names, descriptions, and so on.
- *Locators*: Identify how to locate what the entity is modeling, both physically in the real world, such as a mailing address, or on a technical scale, a position on a computer screen.
- *Values*: Things that quantify something about the entity, such as monetary amounts, counts, dates, and so on.

WOW! eBook
www.wowebook.org

As was true during our entity search, these aren't the only places to look for attributes, but they're just a few common places 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 give you help 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 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, but not always (sometimes you may not be able to guarantee uniqueness, or even guarantee each instance will have a value). For example, here are some common examples of good identifiers:

- *For people*: Social Security numbers (in the United States), full names (not generally a perfect computing 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 number assigned for tracking purposes.
- *For books*: The ISBN number (titles definitely aren't unique, not even always by author).
- *For products*: Product numbers for a particular manufacturer (product names aren't unique), Universal Product Codes, etc.
- *For companies that clients deal with*: These are commonly assigned a customer/client number for tracking that probably wouldn't change even if the client changed names.
- *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.

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 a very important 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 personally use them as keys (and if you know two people named Louis Davidson, heaven help you, you would morph the name to be Louis Davidson the author, or Louis Davidson that other guy who isn't the author), but in a database, doing so becomes problematic. For example, there are many thousands, if not millions, of people named John Smith are out there! For these cases, you may want to identify in the documentation what I call "likely uniqueness."

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, so that while you can't enforce uniqueness, you can use this to let the UI 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. Usually, the process will include not only the given name, but the address, phone number, e-mail address, etc. to start to increase the probability of a match. (In Chapter 8, we will discuss the different ways we can implement uniqueness criteria; for now, it is important to contemplate and document the cases.)

WOW! eBook
www.wowebook.org

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 silly and lead to a lot of confusion. 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 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 in the width of a normal document or form:

```
123456789012345678901234567890123456789012345678901234567890
```

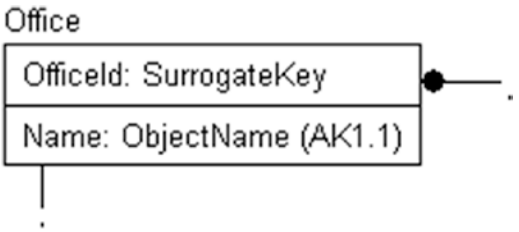


Figure 4-18. *Added the Name attribute to the Office entity*

The actual default length can easily be changed. That is the point of using domains. It may also be clear that the `Office` name is actually a domain of its own.

■ **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.

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 name, unless you are modeling a dentist/carpentry office for Moe, Larry, and Curly.

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. 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).

WOW! eBook
www.wowebook.org

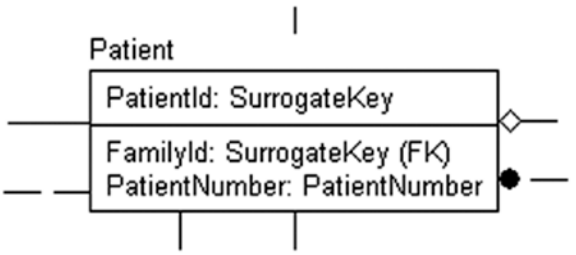


Figure 4-19. *Added the PatientNumber attribute to the Patient entity*

■ **Note** I used the name `PatientNumber` in this entity even though it includes the name of the table as a suffix (something I previously suggested 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 into the customers actual business practices.

A common contra-positive to the 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 tubes of toothpaste? And does the customer 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 some implant that the dentist may use. 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 implants, there are almost certainly serial numbers. Perhaps for medicines like narcotics, we might require a label be printed with a code and maintained for every bottle that is distributed. 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.

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).

WOW! eBook
www.wowebook.org

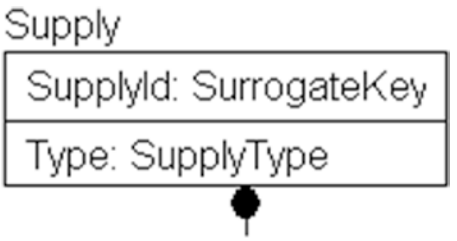


Figure 4-20. *Added the Type attribute to the Supply entity*

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 for a conversation.
- *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****.*

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. A building 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 the `Office` entity after adding the `Address` attribute:

WOW! eBook
www.wowebook.org

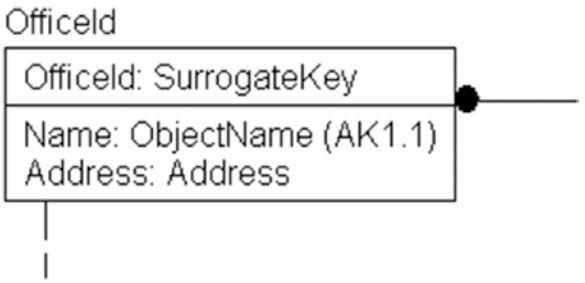


Figure 4-21. *Added an Address attribute to the Office entity*

Each office can have only one address that identifies its location, so the `Address` attribute 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. Don't get hung up on the fact that you know addresses are more complex than a simple value…how you end up implementing addresses is beyond the scope of the conversation. We will discuss the details of how the address is implemented more later in the process, depending on how addresses will be used by the user.

Immovable places aren't the only things you can locate. A 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 (consider how companies may want to track physical assets, like taxi cabs, tools, and so on). 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.

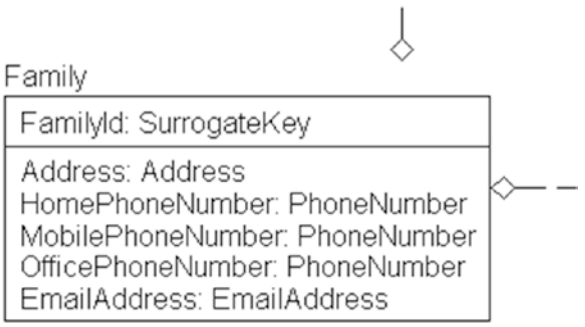


Figure 4-22. *Added location-specific attributes to the Family entity*

WOW! eBook
www.wowebook.org

The same is found for the patients, as shown in Figure 4-23.

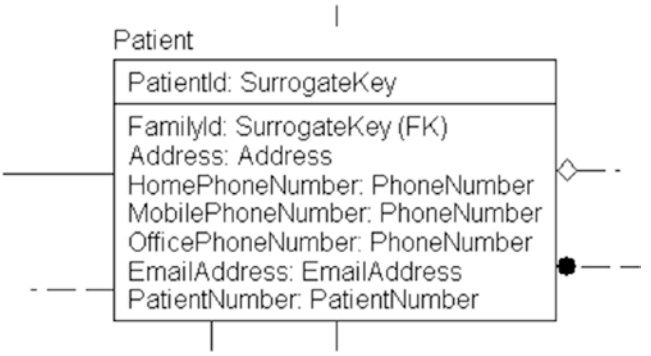


Figure 4-23. *Added location-specific attributes to the Patient entity*

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 the physical database you implement needs to fit a certain mold of being a scalar or fixed vector and nothing else. The normalization process, which will be covered in Chapter 5, completes the process of shaping all of the attributes into the proper shape for implementation in our relational database.

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 (which is not as funny when Rapunzel isn't doing it in *Tangled*!).

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 (e.g., 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, dimensions 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 10. It would not be a hard and fast rule, but it would be a sanity check, because most people don't have 10 dependents (well, most sane people, before, or certainly not after!). You might also then specify a top limit that is not supported if the law limited it. 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 DDL, but they have to be implemented somewhere, or users can and will put

WOW! eBook
www.wowebook.org

in whatever they feel like at the time). It is nice to establish a sanity value, so one doesn't accidentally type 100 when meaning 10 and get audited.

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.

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 them over and over as well as when you are running through tests until you are happy with your design and move past the logical model and start to produce a physical one.

A List of Entities, Attributes, and Domains

Figure 4-24 shows the logical graphical model as it stands now and Table 4-3 lists the entities, along with descriptions and column domains. The attributes of an entity are indented within the Entity/Attribute column (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.

WOW! eBook
www.wowebook.org

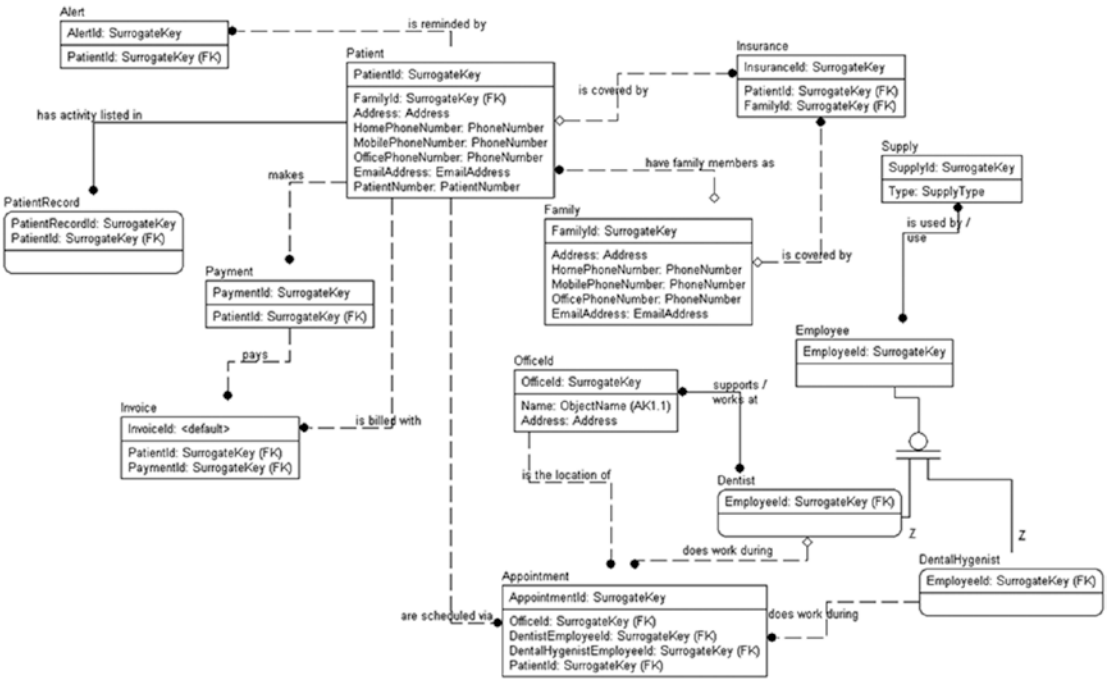


Figure 4-24. *Graphical model of the patient system so far*

WOW! eBook
www.wowebook.org

Table 4-3 lists a subset of the descriptive metadata.

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?*)

(*continued*)

WOW! eBook
www.wowebook.org

Table 4-3. (*continued*)

Entity/Attribute	Description	Column Description	Column Domain
`Dentist`	Persons who do the most skilled work at the dental office. Several dentists work for the client's practice.		
`DentalHygienist`	People who do the routine 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`	Supplies for the running of the dental operations of the office. Examples given were sample items, such as toothpaste or toothbrushes; plus, there was mention of dental supplies as the supplies that the dentists 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.

WOW! eBook
www.wowebook.org

■ **Tip** Carefully review 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 3000.

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.

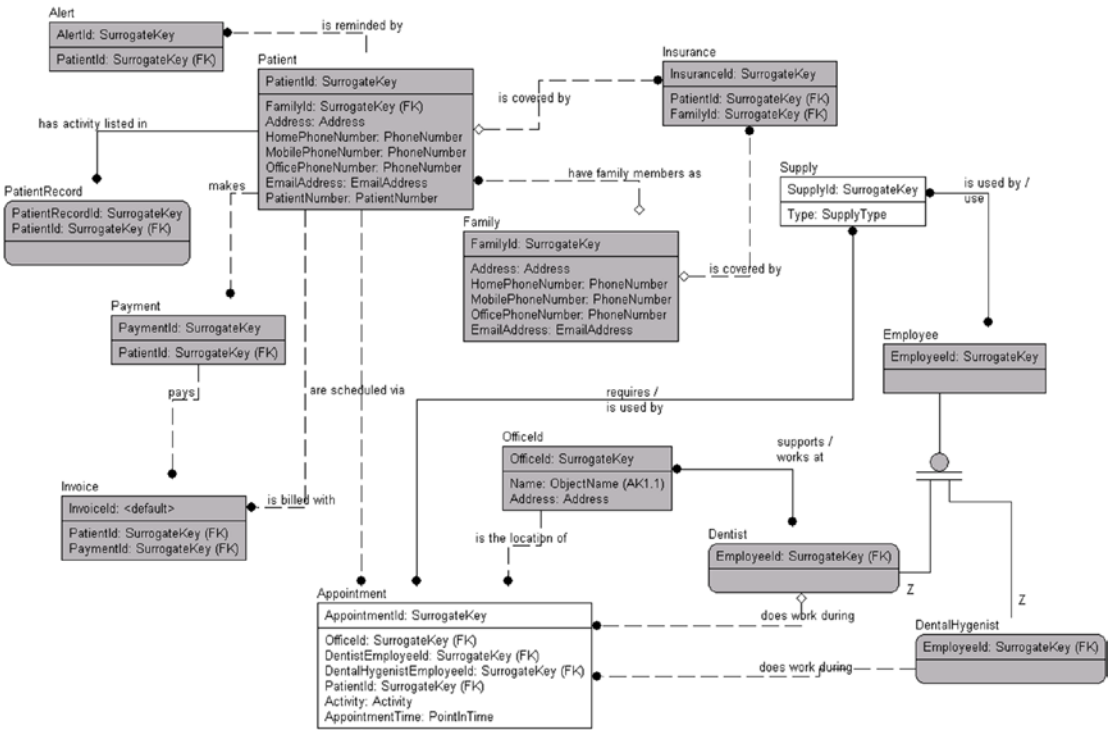


Figure 4-25. *Model with all entities, attributes, and relationships that were found directly in the model*

At this point, the entities and attributes have been defined. Note that nothing has been added to the design that wasn't directly impliable from 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, and would be far closer to what might be implemented (including holes like `Employee` and its subtypes not having any attributes other than a surrogate key placeholder). 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 the parts of the process clearer (and small enough to fit in the book!).

It might also be interesting to note that the document itself is 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 more redundancy in much of the documentation.

WOW! eBook
www.wowebook.org

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 (and too often they are stored only 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. It will often become an argument about why we are discussing such things, but it will pay off later when you are required to write reports and have some frame of reference to know if $100000 is an acceptable value for a patient's charges.

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, but missing business rules can ruin data quality for reporting, or even wreck an 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 (certainly none that is regularly followed by a large number of organizations regularly).

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. (made worse when it is code that you actually wrote.)
- *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 story, 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 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.

WOW! eBook
www.wowebook.org

In our "snippet of notes from the meeting" example, a few business rules need to be documented. 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.*

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` (since most dentists' offices don't work for free). 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 there was a holiday or 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, and as such it might manifest itself as a report, or an alert in the UI.

The specifics of some types of rules will be dealt with later in Chapters 6 and 7, and often throughout the book, as we implement various types of tables and integrity constraints.

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 time the patient fills out a set of forms, then the receptionist asks many of the same questions, and finally, the hygienist and dentist 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. If you are familiar with UML diagrams, these might be called out as a key term.

WOW! eBook
www.wowebook.org

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, sometimes 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 your first 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.
10. Drive safe out there.

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, filling in multiple forms, 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 (if you received the license without passing the exams, for example, that 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 with a certain grade, 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:

> *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?

WOW! eBook
www.wowebook.org

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. Electronic signatures may need to be registered as well. 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, especially any changes made in the database to the patient records."
- *Alert patient*: From "alerting the patients when and where 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.

Finalizing 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 little chance 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.

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 computer system. On the other 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).

WOW! eBook
www.wowebook.org

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 (and will be far more complete than my requirements were in any case). At this point in the design, you need to change direction and begin to add the attributes that come naturally to fill in any gaps. 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.

WOW! eBook
www.wowebook.org

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 these few should make the point clearly enough. 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. And as usual, these new items need approval of the client before implementation.

Testing the Logical Model

Just like the conceptual model, it is very much worth it to again test that everything your requirements required is still possible based on the model you are putting out. This step will help you not miss any detail. We won't spend any more text on this concept, as it is simply doing exactly what we have already done for the conceptual model, with more focus spent on whether we have the attributes to support the stated needs.

If you are any sort of programmer these days, you have heard of test-driven development. The concept is that the earlier you catch defects, the cheaper they are to mitigate. Problems are an order of magnitude (or more) easier to fix in the logical model than the physical model. And once you have a database created, and then code referencing it, forget about fixing some problems. You may just have to work around some of the issues. Plus the tests you do now in words will translate to the tests that are performed in code as well.

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, and certainly not on the first try. The most important thing is to get as much correct as you can and get it in front of the customer to agree on. 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, and 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.

WOW! eBook
www.wowebook.org

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 client's. 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 a hydrogen-filled, thermite-coated dirigible.

■ **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 your 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 15th 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. The way in which I present the process in this book is intended to encourage you to follow a reasonably linear process rather than starting out with a design and 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. The following chapters are going to start back down the hill toward producing output without thinking about how we got the requirements.

WOW! eBook
www.wowebook.org

Summary

In this chapter, I've presented the process of discovering the structures that should eventually make up a simple dental-office database solution. We've waded 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 you 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 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.

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 logical data model will emerge that has most 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 most of the same characteristics and will probably not be so different that even the nontechnical layperson who has to approve your designs will understand it.

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 Chapter 5 to produce portions of data models that demonstrate the many ways you can take these very basic skills and create complex, interesting models.

WOW! eBook
www.wowebook.org

CHAPTER 5

Normalization

I do not like being famous. I like being normal.

—Vince Gill, American country singer

By now, you should have built the conceptual and logical model 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 (even if it probably will), the main point being that it does have to cover the data-related requirements for the system that needs to be built.

Now, we come to the most hyped (and sometimes most despised) topic in all relational databasedom: *normalization*. It is where the theory meets reality, and we translate the model from something loosely structured to something that is very structured to follow a certain pattern. 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 has been designed to work with. Once you are done with the process, working with the data will be more natural using SQL.

SQL is a language designed to work with sets of 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 entities and attributes into atomic units; that is, break them 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 and seasoned veterans alike because one must resist the temptation to go too far. One analogy is to consider breaking down a water molecule into its constituent parts. It is possible to split hydrogen and oxygen from water into their own forms, and we can do that safely (and put them back together as well) without changing the nature of the atoms, and this may even be desirable for certain applications. However, if you split the hydrogen atom in half to get to the particles that make it up, problems will occur, like a huge crater. A similar concern exists for tables and columns in a database. At the correct atomic level, using the database will be natural, but at the wrong level of atomicity (too much or too little), you will find yourself struggling against the design in far worse ways than having to write a bit more code than you initially expected.

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 searches, reports, and consistent results from their data being stored, and it will no doubt take you quite a few tries before you start realizing just how true this is.

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

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_5

WOW! eBook
www.wowebook.org

present isn't numbered; it's named for the people who devised it, and encompasses two of the numbered forms.) 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).

In this chapter, I will present 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, describe the programming anomalies they help you avoid, and identify 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 preprogramming design, but the point of normalization is to transform the entity model we have started with into something that is implementable as tables. So it is very important to start thinking as a programmer so you can reconcile *why* having data in a given normal form can make the tables easier to work with in SQL (otherwise it tends to just look like more work for the sake of some arcane practices of old). 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 until every entity/table expresses one and only one concept. The process continues until we produce a model such that, in practical terms, every table in the database will represent one thing and every column in each table describes that thing the table is modeling. This will become more apparent throughout the chapter as I work through the different normal forms.

■ **Note** There is a theorem called "Heath's Theorem" that can explain the process of normalization in more mathematical terms. You can read about it in *Database Systems: A Pragmatic Approach, Second Edition*, by Elvis Foster (www.apress.com/9781484211922), or in Professor William Perrizo's online notes for a college database design class, posted at www.cs.ndsu.nodak.edu/~perrizo/classes/765/nor.html.

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 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 few designs, and even fewer implementations, meet any of the normal forms perfectly, and just because you can't meet one criteria doesn't mean you should chuck the more advanced criteria. Just like breaking down and having a donut on a diet doesn't mean you should go ahead and eat the entire box, 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. 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.

WOW! eBook
www.wowebook.org

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 were to ignore the rest of this book, it would be minimally important to understand how the relational engine wants the data shaped. Recall from Chapter 1 that the first two of Codd's rules are the basis for the definition of a table. The first states that data is to be represented only by values in tables, and the second states that any piece of data in a relational database is guaranteed to be logically accessible by knowing its table name, primary key value, and column name. This combination of rules sets forth the requirements that

- All columns must be atomic; that is, only a single value is 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 a value would not be extended to implement arrays, or, even worse, that position-based fields having multiple values would 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.

Violations from the three presented criteria 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 where we desire data to be modified at any location, by any user, often by many users simultaneously. Even data warehousing databases generally follow First Normal Form in order to make queries work with the engine more optimally. The later normal forms are less applicable to data warehousing situations 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 data value like `'1, 2, 3, 6, 7'`. This likely represents five separate values. It may not, and whether or not it is five different values will be up to the context of the customer's needs (the hardest part of normalization is that you have to be able to separate what something looks like versus what it means to the users).

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 previously mentioned— `'1, 2, 3, 6, 7'`— if the list is always treated as a single value by the client and in turn, in the SQL code, it might be acceptable to store the value in a single column. However, if you might need to deal with the value `3` individually, the value is 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.

WOW! eBook
www.wowebook.org

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 only makes sense as a single value and 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 testing atomicity, the test of reasonability is left up to the designer. However, 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 your requirements!). And why stop at paragraphs? Why not sentences, words, letters, or even bits that make up the characters? Each of those breakdowns may actually make sense in some context, so understanding the context is key to normalization.

■ **Note** While you can normalize pretty much any data structure, there are other technologies that can be used to implement a database, such as Hadoop, NoSQL, DocumentDB, etc., that, even at this point in the process, may actually make more sense for your implementation. Just because all you own is a hammer doesn't make every problem solvable with a nail.

As examples, consider some of the common locations where violations of this rule of First Normal Form often can be found:

1. E-mail addresses
2. Names
3. 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 to 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 to 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

WOW! eBook
www.wowebook.org

more than one e-mail address in a single column and leads to difficult utilization in your Transact-SQL. In Chapter 10, it will become more 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.

If users are allowed to have more than one e-mail address, the value of an e-mail 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).

■ **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 we will discuss later, but good enough for this discussion):

```
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 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, and while it is pretty messy compared to the proper solution (a table with one row per e-mail address for each person), it is very similar to what one might write in a language like C# one row at a time. However, there are problems first with dealing with the true complex nature of data formats (for example, `"email;"@domain.com` is, in fact, a valid e-mail address based on the e-mail standard!; see www.lifewire.com/elements-of-email-address-1166413), but the biggest issue is going to be in how the relational engine works with the data. Comparisons on entire column values (or at least partial values that include the leading characters) can be optimized well by the engine using indexes. A good test of what is correct is to look for data with characters for formatting data that has no meaning to the user. Format data for use, not for storage. It is easy to format data with the UI or even with SQL.

Consider other common operations, such as counting how many distinct e-mail addresses you have. With multiple e-mail addresses inline, using SQL to get this information is painful at best. But, as mentioned, you should 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
```


WOW! eBook
www.wowebook.org

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
```

Now, an easy query determines how many e-mail addresses there are per person:

```
SELECT PersonId, COUNT(*) AS EmailAddressCount
FROM   PersonEmailAddress
GROUP BY PersonId;
```

And the previous update we wrote is simply written as

```
UPDATE PersonEmailAddress
SET    EmailAddress = 'family@bull.com'
WHERE  EmailAddress = 'tay@bull.com';
```

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 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).

WOW! eBook
www.wowebook.org

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 our clients are 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, which is actually worse than what you started with!

```
PersonId       EmailAddress
============== ================================================
0001003        tay@bull.com
0001003        tay@bull.com;taybull@hotmail.com;tbull@gmail.com
0001003        tbull@gmail.com
0003020        norma@liser.com
```

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: first and last names when greeting someone we don't know, first name 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 name and last name if you needed them on occasion. Parsing seems simple, assuming every name is formatted with precisely one first name, one middle name, and one last name. Add in names that have even slightly more complexity though, and parsing becomes a nightmare.

Consider the following list of names:

```
PersonId    FullName
=========== ---------------------------
00202000    R. Lee Ermey
02300000    John Ratzenberger
03230021    Javier Fernandez Pena
```


WOW! eBook
www.wowebook.org

This "one, big column" initially seems to save a lot of formatting work, but it has a lot of drawbacks. The problem with this approach is that it is really hard to figure out what the first and last names 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  FullName
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  FullName
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
```

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 minimally provide the user with first, middle, and last name fields to fill out. Then allow the user to decide which parts 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 also an artist, just not up to Leonardo's quality) considers Da as his middle name.

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, 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 Ratzenbergers (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.

WOW! eBook
www.wowebook.org

For example, you might store versions of a person's name to be used in greeting the person (`GreetingName`), or to reflect how 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 to normalization? Not at all. The name used to talk to the person might be Q-dog, and the given name Leonard. Duplication of data is only going to be an issue when there can be no exceptions. In the end, judging the usage is not our job; our job is to model it to support the clients utilization. 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. So we will want to document the dependencies that naturally exist, even if the dependencies are not constant and can be overridden.

Note, however, that no matter how well you think this design works, there are still problems with it if getting everyone's name exactly correct exists. While it is common in the United States for most people to have three-part names, it is not the custom for everyone here, and is certainly not the case for other nationalities. What to do if a person has ten names? Stuff eight of the names in the `MiddleName` column? If you want to get it perfectly right (as you may need to do if you are doing certain types of applications), you need to allow for as many words as might ever be necessary (note that we also get some level of metadata now about each part of the name). Of course, this sort of solution is very rare, and usually unneeded for almost all implementations, but if you needed to store an unlimited number of name parts for a person, this would be the best way to go:

```
PersonId    NamePart    Type         Sequence
=========== ----------- ------------ --------
00202000    R.          First        1
00202000    Lee         Middle       2
03230021    Javier      lastname     3
03230021    Fernandez   First        1
00202000    Ermey       lastname     2
```

■ **Tip** Names are an extremely important part of many customer systems. 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

U.S. 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 U.S. telephone numbers in a database. The decision on how to handle this situation is usually based on how often the users store international phone numbers, because it would be a hard task to build a table or set of tables to handle every possible phone format, and certainly more difficult than most clients will support paying you for.

WOW! eBook
www.wowebook.org

For a North American Numbering Plan (NANP) phone number (America, Canada, and several other countries), you can represent the standard phone number with three different columns for each of the three following parts, AAA-EEE-NNNN (the country is always 1):

- *AAA (Area code)*: Indicates a calling area located within a region
- *EEE (Exchange)*: Indicates a set of numbers within an area code
- *NNNN (Number)*: Number used to make individual phone numbers unique

Whether you make three columns for phone numbers is a tricky decision. If every phone number fits this format because you only permit calling to numbers in the United States and Canada, having three columns to represent each number would be a great solution. You might 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 did not. Should you 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.

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 will not be on time. In addition, 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 messy and certainly not the best performer. Usually, when an area code splits, it is 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 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. Consider even a slight change, as in the following values that have an extra space character thrown in for your programming amusement:

```
PhoneNumber
==============
 615-555-4534
615- 434-2333
```


WOW! eBook
www.wowebook.org

You are not going to be able to deal with this data simply, because neither of these rows would be updated by the previous UPDATE statement.

Changing the area code is much easier if all values are stored in single, atomic, strongly domained containers, as shown here (each container only allowing the exact number of characters that the real world allows, which in the NANP phone number is three, three, and four characters respectively):

```
AreaCode  Exchange   PhoneNumber
========= ========== ==============
615       555        4534
615       434        2333
```

Now, updating the area code takes a single, easy-to-follow SQL statement such as

```
UPDATE PhoneNumber
SET    AreaCode = '423'
WHERE  AreaCode = '615'
  AND  Exchange IN ('232','323',...,'989');
```

How you represent phone numbers in a database is a case-by-case decision driven by requirements. 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 this 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 other major countries don't use the same format as in the United States and Canada. In addition, they all have the same sorts of telephone number concerns as we do with the massive proliferation of telephone number–addressed 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 how the numbers can be accessed 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, on occasion I may 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

WOW! eBook
www.wowebook.org

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, such as on the area code or partial phone numbers, and design accordingly.

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 values. There are two interpretations of this:

- Tables must have a fixed number of columns.
- Tables need to 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 strictly possible with a relational database table. (Note that, internally, the storage of data in the storage engine may 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 the engine.)

The second interpretation, however, is more about how you use the tables you create. As an example, if you are building a table that stores a person's name and you have a column for the first name, then all rows must have only one first 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! That a value is already known is not the question, but rather the existence of a value that you might know.

The most obvious violation of this rule is where people make up several columns to hold multiple values of the same type. An example is a table that has 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       23.32         120.23
0000230003       Fred's Cat Shop   200.23       NULL          NULL
```

Each column represents one payment, which makes things easy for a programmer to make a screen, but consider how we might add the next payment for Fred's Cat Shop. We might use some SQL code along these lines (we could do something that is simpler looking, but it would do exactly the same logically):

```
UPDATE 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';
```


WOW! eBook
www.wowebook.org

Of course, if there were already three payments, you would not have made any changes at all, losing the change you expected to make, and not registering the update. 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 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."

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 any 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 details like name. The second table holds payments by customer.

```
CustomerId       Name
================ ----------------
0000002323       Joe's Fish Market
0000230003       Fred's Cat Shop

CustomerId       Amount       Date
================ ------------ --------------
0000002323       100.03       2015-08-01
0000002323       23.32        2015-09-12
0000002323       120.23       2015-10-04
0000230003       200.23       2015-12-01
```

Now adding a payment to the table is simple in one statement:

```
INSERT CustomerPayment (CustomerId, Amount, Date)
VALUES ('0000230003',1000,'2016-02-15');
```

As in earlier examples, notice that I was able to add an additional column of information about each payment with relative ease—the date each payment was made. What is worse is that, initially, the design may start out with a column set like `Payment1`, `Payment2`, but it is not uncommon to end up with a table that looks like this, with repeating groups of columns about the already repeated columns:

```
CustomerId       Payment1     Payment1Date  Payment1MadeBy Payment1ReturnedCheck Payment2...
================ ------------ ------------- -------------- --------------------- ---------
0000002323       100.03       2015-08-01    Spouse         NULL                  NULL
0000230003       200.23       2015-12-01    Self           NULL                  NULL
```

It wasn't that long ago that I actually had to implement such a design to deal with the status and purpose of a set of columns that represented multiple e-mail addresses for a customer, because the original design I inherited already had columns `emailAddress1`, `emailAddress2`, and `emailAddress3`.

In the properly designed table, you could also easily 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

WOW! eBook
www.wowebook.org

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 agilely (within reason.)

Now, with payments each having their own row, adding a payment would be as simple as adding another new row to the `Payment` table:

```
INSERT Payment (CustomerId, Amount, Date)
VALUES ('000002324', $300.00, '2016-01-01');
```

Often, the user will want a number to say this is the 1st payment, then the 2nd, etc. This could be a column that is manually filled in by the customer, but ideally you could calculate the payment number from previous payments, which is far easier to do using a set-based SQL statement. Of course, 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 the desires of the business whose design you are trying to implement.

Beyond allowing you to add data naturally, designing row-wise rather than repeating groups 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? The extra information about the time of the payment, the check number, etc. would likely clear this up, too.

If your requirements really only allow three payments, it is easy enough to specify and then implement a constraint on cardinality of the table of payments. As discussed 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 (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 the bounds of the current portion of the database design process.

■ **Caution** Another common (mildly horrifying) 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.

WOW! eBook
www.wowebook.org

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% of the time, it usually is very possible. An example is a situation where you cannot tell the physical items apart, such as perhaps cans of corn (or bags of Legos, for that matter). You cannot tell two cans of corn apart based on their packaging, 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. Generally speaking though, if you can't tell two items apart in the real world, don't endeavor to make them distinguishable in your database. It will be perfectly acceptable to assign a key to a shelf of corn along with the number of cans in inventory.

The goal in all cases is to find the lowest granular level where a user may want to distinguish between one thing and another. For example, in retail, it is common to allow returns of items. If a person wants to return a $10 DVD, the retailer will make sure that the item being returned is simply the same thing that the person purchased. On the other hand, if the person has purchased a $20,000 diamond ring, there will likely be a serial number to make sure that it is the same ring, and not a ring of lesser value (or even a fake).

Often, database designers (such as myself) are keen to automatically add an artificial key value to their table to distinguish between items, using a GUID or an integer, but as discussed in Chapter 1, adding an artificial key alone 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. 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. 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.

Another common approach that can be concerning 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 ideal. Conversely, simply tossing on a date and time value to force uniqueness is worse than just adding a random number or GUID on the row when time is not part of what identifies the item being modeled. This is the blue Ford C-Max I purchased on December 22, 2014 at 12:23 PM…or was it the 23rd at 12:24 PM? What if the client registered the purchase twice, at different times? Or you actually purchased two similar vehicles? Either way would be confusing! Ideally in that case, the client would use the vehicle identification number (VIN) as the key, which is guaranteed to be unique. (Though strictly speaking, using the VIN does violate the atomic principal since it loads multiple bits of information in that single value. Smart keys, as they are called, are useful for humans but should not be the sole source of data, like what color is the vehicle, based on the rules we have already stated.)

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
```

Taken as presented, there is no obvious clue as to which of these rows represents 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 at a certain school. 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 (possibly incorrect) interpretations.

WOW! eBook
www.wowebook.org

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   SQL Tacklebox  Simple Talk     Rodney
```

The users go about life, entering data as needed. Nevertheless, 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 (this is a contrived example, but I have heard from more than one person that this has occurred in real databases):

```
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 multiple formats of data, 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 they 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   SQL Tacklebox  Simple Talk
333333333   Indexing       Microsoft
444444444   DMV Book       Simple Talk

BookISBN    Author
=========== =========
111111111   Louis
222222222   Rodney
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 uses 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 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 make software design such a delightful task at times.

WOW! eBook
www.wowebook.org

■ **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 a database of data 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. They are 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 can be found in correct columns. So you need not go too far. As mentioned earlier, 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. In essence, this clue is 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. And you may want to record a third item of ID in case one that is provided is not valid…always must 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 something more like `CoordinateX` and `CoordinateY` would likely be better column names).

WOW! eBook
www.wowebook.org

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 even this is not a given in many cases). However, all too often the key will simply be a GUID or a sequence/identity-based value.

It might seem like I am picking on sequential numbers and GUIDs, and for good reason: I am. While I will usually suggest you use surrogate keys in your designs, too often people use them incorrectly and forget that such values have no meaning to the user. So if Customer 1 could be the same as Customer 2, you are not doing it right. 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`.

Three normal forms 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 BCNF because it encompasses the other forms, and it is the clearest 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 common norm of using a surrogate key for most primary keys, BCNF is a far better definition of how a properly designed database ought to 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.

WOW! eBook
www.wowebook.org

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 and start adding the Y table, this appearance of determinism can 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 functionally 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 for a row, 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 without 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   SQL Tacklebox  Simple Talk     England
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. Therefore, if you needed to change the publisher, you would also need to change the publisher location in multiple rows.

WOW! eBook
www.wowebook.org

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   SQL Tacklebox  Simple Talk
444444444   DMV Book       Simple Talk

Publisher     PublisherLocation
============= ---------------------
Apress        California
Simple Talk   London
```

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.

Of course, there is always a caveat. Consider the following table of data:

```
BookISBN    BookTitle      PublisherName   PublisherLocation
=========== -------------  --------------- ------------------
111111111   Normalization  Apress          California
222222222   SQL Tacklebox  Simple Talk     New York
444444444   DMV Book       Simple Talk     London
```

Now Simple Talk has two locations. Wrong? Or a different meaning that we expected? If the column's true meaning is: "Location Of Publisher At Time Of Book Publishing," then (other than a truly poorly named column) the design isn't wrong. Or perhaps Apress has multiple offices and that is what is being recorded? So it is important to understand what is being designed, to name columns correctly, and to document that purpose of the column as well to make sure meaning is not lost.

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 (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 that 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
```


WOW! eBook
www.wowebook.org

The key of `Driver` plus `VehicleStyle` 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. If it is the height of the car, it is still wrong!
- `EyeColor`: Clearly, this references the driver only, unless we rent Pixar car models. Either way it not referencing the combination of `Driver` and `VehicleStyle`
- `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 actually is correct in this table.

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   Car Style        DesiredModelLevel
======== ================ ------------------
Louis    CUV              Premium
Louis    Sedan            Standard
Ted      Coupe            Performance
```

Finally, we need one table to define the types of car styles available (and I will add a column to give an example model of the style of car):

```
Car Style        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 have many, many rows; the table assigning drivers to car styles would have very few rows and would be thin (have a small number of columns); and the car style table would be very small in number of rows and columns. The savings from not repeating so much data will more than overcome the overhead of doing joins on reasonably tuned 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.

WOW! eBook
www.wowebook.org

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 deals with the primary key, but BCNF expands it to include all defined keys.) When we have completed our design, and it meets the standards of BCNF, every possible key will have been designed and enforced.

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
```

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 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 the entire 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? We would end up with two tables that look like this:

WOW! eBook
www.wowebook.org

```
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 could only use data from a given domain that is defined by the vehicle type table. Note too that the two 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 semantically the same as using 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 `employee` table is in proper BCNF, you check out the columns and you come to `driversLicenseNumber` and `driversLicenseStateCode`. Does an employee have a `driversLicenseStateCode`? Not exactly, but a driver's

WOW! eBook
www.wowebook.org

license does, and a person may have a driver's license. 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? Absolutely.

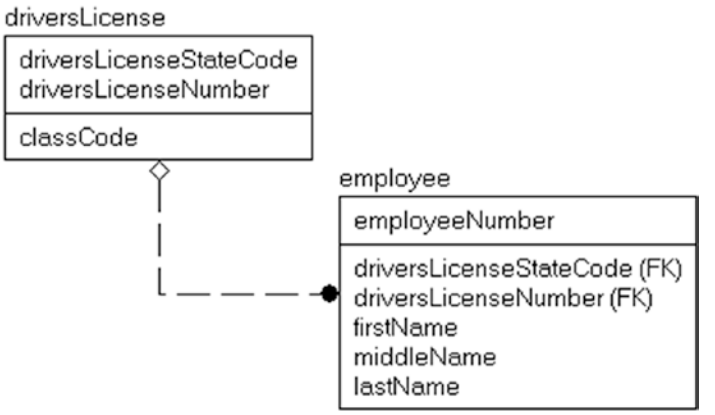


Figure 5-1. *driversLicense and employee tables with natural keys*

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.

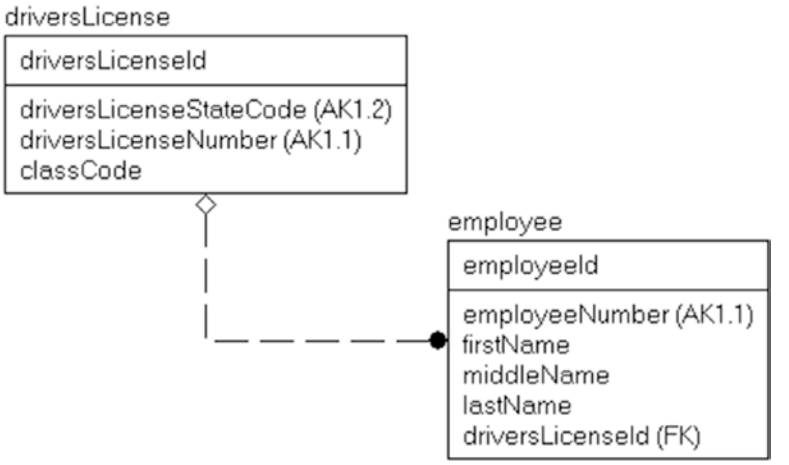


Figure 5-2. *driversLicense and employee tables with surrogate keys*

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 normalization issues. For example, take the addition to the model shown in Figure 5-3, made by designers who weren't wearing their name-brand thinking caps.

WOW! eBook
www.wowebook.org

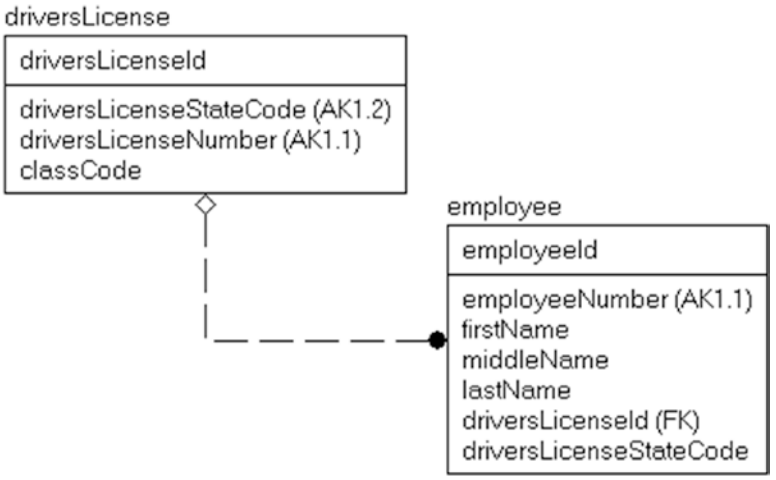


Figure 5-3. *driversLicense and employee tables with improper normalization*

The users wanted to know the state code from the driver's license for the employer, so the programmer simply added it to 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 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 joins 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.

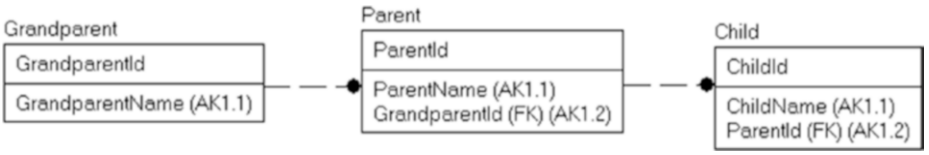


Figure 5-4. *Tables chained to show key migration*

WOW! eBook
www.wowebook.org

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 table, 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.

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. 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          1232322000000000323
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, be careful if you see data like this. You have to question whether or not the InvoiceAmount is a true calculation, or a value that has to be balanced against. The value of 100 may be manually set as a check to make sure that no items are changed on the line items. The requirements must always be your guide when deciding what is and isn't wrong in normalizations, and all of database/software design.

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 really bad one!), not to mention that prices can change. So as in pretty much everything in this book, design for what the requirements are, and name and document everything so well that when people (or programmers) look at the data, they will not be confused as to whether duplicated data is there on purpose or you were just clueless when you designed the table.

WOW! eBook
www.wowebook.org

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

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 (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 that your design isn't in BCNF. 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 "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.

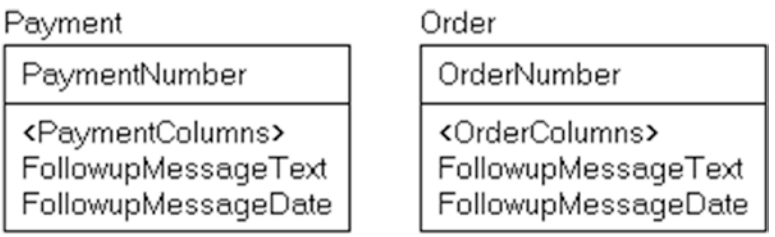


Figure 5-5. *Payment and Order with errant Followup columns*

The tables in Figure 5-5 are a glowing example of information 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.

WOW! eBook
www.wowebook.org

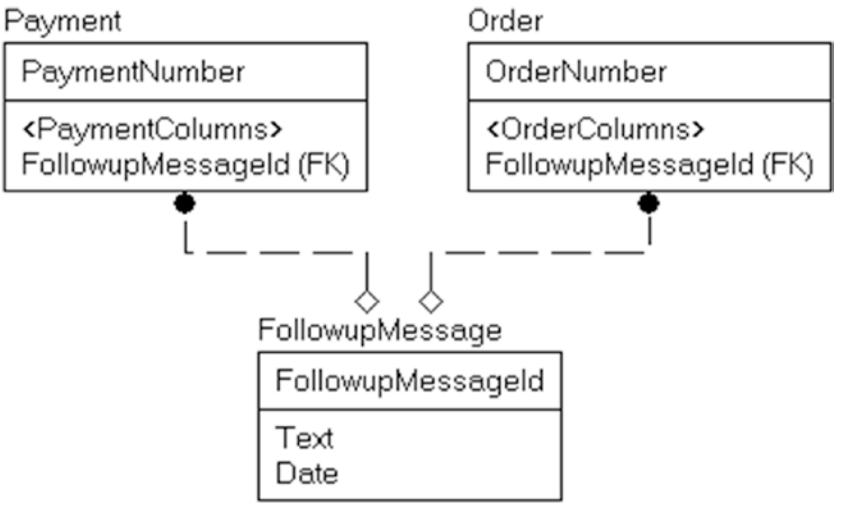


Figure 5-6. *Payment and Order with added Followup object*

This design in Figure 5-6 allows you to tag a given message to multiple payments, orders, and can use that information to form more information (for customers who made an order and were sent a follow-up message in N days, they were P% more likely to purchase…).

Repeating Groups of Data

More difficult to recognize are the repeating groups of data, often because the names given to columns won't be quite so straightforward as you may like. 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   SQL Tacklebox  Simple Talk     London
444444444   DMV Book       Simple Talk     London
```

The repeating values (`Simple Talk` and `London`) are a clear example of something that is likely amiss. It isn't a guarantee, of course, since (as mentioned earlier) it may be the publisher's location at the time of the order. It can be beneficial to use a data-profiling tool and look for these dependencies not as absolutes, but as percentages, because if there are a million rows in this table, you are very likely to have some bad data no matter how good the code is that manipulates this data. In essence, you 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.

WOW! eBook
www.wowebook.org

Summary Data

One of the most common violations of BCNF that might not seem obvious is summary data. 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 usually there is very little place for it. Not only is summary data not functionally dependent on nonkey columns, it's dependent on columns from a different table altogether. Summary data should be reserved either for dealing with extreme performance tuning, much later in the database design process, or ideally for 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.

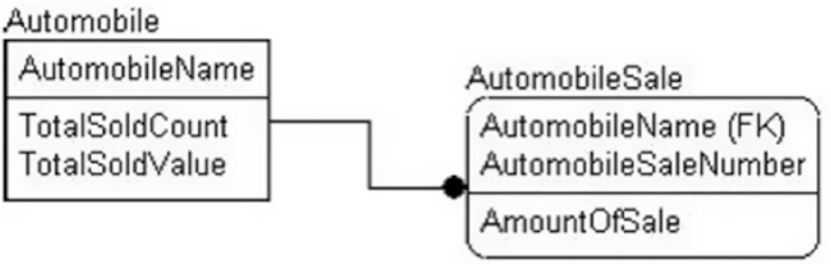


Figure 5-7. *The auto dealer submodel*

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. Just to constantly remind you that nothing is necessarily 100% wrong, if you included a `PreviouslySoldCount` on `Automobile` that contained sales that had been archived off, it would not be a normalization issue. A common phrase that is mentioned around this is "single source of the truth." The old saying goes, "A person with one watch knows what time it is, and a person with two has no idea," as you can never get two watches to match, even with electronics these days. Best to calculate the values as much as possible, until there is absolutely zero chance of the data changing.

The point is that before you start implementing, including summary data on the model isn't generally 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.

WOW! eBook
www.wowebook.org

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 `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
```

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 project I recently was working on, half of the users used the latter addresses as historic of old e-mail addresses and the other half as a backup e-mail for contacting the customer. For the historic 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. However, 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.

WOW! eBook
www.wowebook.org

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 the product 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

Assuming you (A) have done some work with databases before getting this deep in the book and (B) haven't been completely self-taught while living underneath 100,000 pounds of granite, you may have wondered why this chapter on normalization did not end with the last section. 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 fourth and Fifth Normal Forms now focus on the relationship between key columns to make sure the keys truly do have a singular meaning. If all of the natural composite keys for your tables have no more than two independent key columns, you are guaranteed to be in Fourth and Fifth Normal Forms 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% of all databases had issues with Fourth Normal Form. And back in 1992, people spent a lot of time doing design for months on end, 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, meaning you must understand your requirements to know if there is an issue or not. The same table 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.

WOW! eBook
www.wowebook.org

Fourth Normal Form: Independent Multivalued Dependencies

Fourth Normal Form deals with multivalued dependencies. When we discussed dependencies in the previous sections, we discussed the case where `fn(x) = y`, where both `x` and `y` were scalar values. For a multivalued dependency, the `y` value can be an array of values. So `fn(key) = (nonkey1, nonkey2, . . . , nonkeyN)` 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.

As an example: recall a previous example table we used for a rental car database:

```
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 currently configured to drive the CUV style (and Louis is the only driver configured for Sedan, Ted is the only driver configured for Coupe, etc.). 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 (or more) 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 type of class that is assigned to use a certain book (in this simple example, consider a class to be a class taught at a given time in a location for a topic):

```
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
```

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 of 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, 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. You can see from the sample data too that we have duplicated data in that the `Normalization` class has the book recorded twice. 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, the first recording the trainers for a

WOW! eBook
www.wowebook.org

class, and the other essentially representing the Class entity with a Book attribute (note the key is now the Class, and not Class and Book). These two tables are equivalent to the initial table with the following three-part key:

```
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, though this may not always be the case. For example, if the class could use multiple books, you might not be able to get back the original table from the joins, as one of the issues here is that, because there are multiple meanings in the key, there is the likelihood of redundant data. Hence, you could end up with the following data:

```
Trainer    Class          Book
========== ============== =========================
Louis      Normalization  DB Design & Implementation
Chuck      Normalization  AnotherBook on DB Design
Fred       Implementation DB Design & Implementation
Fred       Golf           Topics for the Non-Technical
```

Since Normalization represents one class, we would break down the tables into the following tables (where the second table is no longer the Class entity, but a table representing the books for a class, and the key is now on Class and Book):

```
Class           Trainer
=============== ==============
Normalization   Louis
Normalization   Chuck
Implementation  Fred
Golf            Fred

Class          Book
============== =========================
Normalization  DB Design & Implementation
Normalization  AnotherBook on DB Design
Implementation DB Design & Implementation
Golf           Topics for the Non Technical
```


WOW! eBook
www.wowebook.org

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:

```
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.

Of considerable concern from an architectural standpoint is how this plays out when using surrogate keys. Most often, a designer would have noted that `VehicleStyle` and `VehicleBrand` are related and thus would have created the following:

```
VehicleStyleBrandId VehicleStyle                VehicleBrand
=================== =========================== ================
1                   Station Wagon               Ford
2                   Sedan                       Hyundai
3                   Coupe                       Chevrolet
```

Which would be related to:

```
Driver             VehicleStyleBrandId
================== ===================
Louis              1
Louis              2
Ted                3
```

It is important to understand that this table has the same key (`Driver, VehicleStyle, VehicleBrand`) as the original table, and the same checks ought to be carried out no matter how the table is defined. Secondly, the determination that `VehicleStyle` and `VehicleBrand` are related to one another is up to interpretation of business rules. It might be that you independently pick brands and styles that are matched to inventory, even if they are not logical (`Ford Motor Company, Motorcycle`, for example). This is the primary reason why these rules are very business rule oriented. Slight variations of cardinality from the user requirements can make the same table design right or wrong (and consequently end up driving your users bananas when they expect one interpretation and get a different one to work with).

WOW! eBook
www.wowebook.org

Fifth Normal Form

Fifth Normal Form is a general rule that breaks out any data redundancy that has not specifically been culled out by Fourth Normal Form. Like Fourth Normal Form, Fifth Normal Form also deals with the relationship between key columns. The idea is that if you can break a table with three (or more) dependent 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 seldom 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 is best to break down any existing ternary (or greater) relationship into binary relationships if 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`.

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 repeatedly.

At this point, you probably now see why 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

WOW! eBook
www.wowebook.org

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 I created:

```
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 cars), the table just presented might be interpreted as

- Louis is willing to drive Ford sports cars, Hyundai sedans, and Volvo station wagons and sedans.
- Ted is willing to drive a Chevrolet coupe.

In this case, the original 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 a dependency is `Driver` to `VehicleStyle` plus `VehicleBrand`. This was the solution we arrived at in the "Fourth Normal Form" section, noting that in most cases, the designer would have easily noted that `VehicleStyle` and `VehicleBrand` would almost certainly be formed in their own table, and the key would be migrated to our table to form a three-part key, either using natural keys or perhaps via a surrogate.

WOW! eBook
www.wowebook.org

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 requirements and reflected in the name of the table:

- This table is 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.
- This 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, only editor Y edits Z's work."

■ **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 BCNF, 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. Obviously, if you can't reconstruct the data that is desirable from 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**

WOW! eBook
www.wowebook.org

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 easy enough to implement. This, of course, introduces risks of data anomalies or even making the data less appropriate for the relational engine. Any additional code written to deal with these anomalies must 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% 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, suppose the user interface in Figure 5-8 was fashioned for a book inventory system.

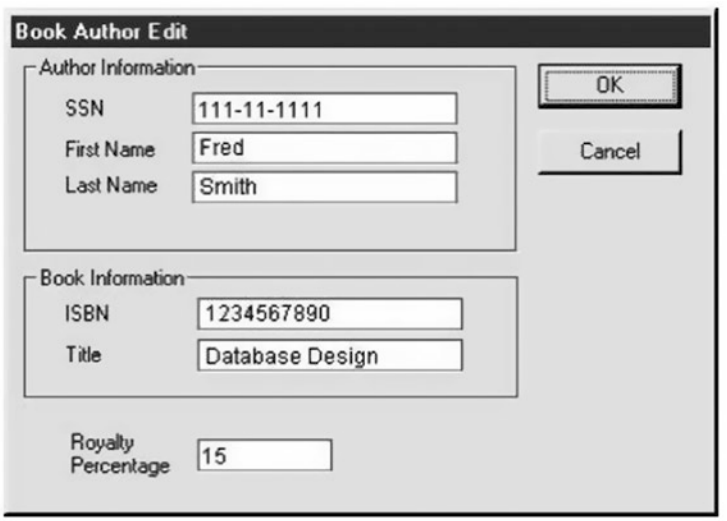


Figure 5-8. *A possible graphical front-end to our example*

WOW! eBook
www.wowebook.org

Does Figure 5-8 represent a bad user interface (aside from my ancient Windows ME–style interface aspects, naturally)? 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 structure should dictate the UI. When the user figures out the problem with expecting a single author for every book, you won't have to change the underlying database design.

■ **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 a large percentage of all books for your client have one author.

UI design and database design are completely separate things. The power of the UI comes with focusing on making the top 80% 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% 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, before a performance issue is actually felt/discovered and found to be non-tunable using available technology.

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 you implement physically what is true logically. For almost all cases, I always advocate waiting until you find a compelling reason to denormalize (such as if some part of your system is failing) before you actually 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 month 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.
>
> *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 after row creation, 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% 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.

WOW! eBook
www.wowebook.org

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:

```
InvoiceNumber  InvoiceDate      InvoiceAmount
============== ---------------- --------------
000000000323   2015-12-23       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   Invoice
GROUP  BY InvoiceNumber, InvoiceAmount
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:

- *Normalization is not an academic process*: It is a programming process. A value like `'a,b,c,d,e'` stored in a single column is not in and of itself incorrect. Only when you understand the context of such a value can you know if it needs one or five columns, or one or five rows to store. The value from the decomposition is to the programming process, and if there is no value, it is not worth doing.
- *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 maximizing utilization of 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.

WOW! eBook
www.wowebook.org

- *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 meaningless 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.

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.

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 and keys are broken down to binary relationships when the decomposition is lossless.

Is it always necessary to go through the steps one at a time in a linear fashion? Definitely not, and after several complex designs you would go pretty batty. 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.
- *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. Basically, to continue with an analogy from the previous chapter, we are just refining the blueprints to get them closer to something we can hand off to an engineer to build. 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!). Now it is time 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).

WOW! eBook
www.wowebook.org

Still not convinced? 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. A side effect of eliminating duplicated data is that less data is stored on disk or in memory, which can be the biggest bottlenecks.
- *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.

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. You know what the client wants, and the client knows that you know what they want (and you have their agreement that you do understand their needs!).
- 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, you broke down these entities and turned them into basic functional relational tables such that every table relays a single meaning. One noun equals one table, pretty much.

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 implementing some designs, beginning with simplistic requirements and logical designs and then building SQL Server 2016 objects in reality. (Most examples will work in 2012 and earlier without change. I will note in the text when using specifically new features along with comments in the code as to how to make something work in pre-2016 versions if

WOW! eBook
www.wowebook.org

possible.) In all likelihood, it is probably what you thought you were getting when you first chunked down your hard-earned money for this book (ideally your employer's money for a full-priced edition).

If you don't have SQL Server to play with, this would be a great time go ahead and get it (ideally, the Developer edition on a machine you have complete control over). You can download the Developer edition for free from `www.microsoft.com/en-us/sql-server/sql-server-editions-developers` or you could use any SQL Server system you have access to (just don't do this on a production server unless you have permission). Most everything done in this book will work on all editions of SQL Server. Installing pretty much with the defaults will be adequate if you are installing this on the same computer where you will be running the toolset. You can install SQL Server Management Studio (the client for managing SQL Server databases) from `msdn.microsoft.com/en-us/library/hh213248.aspx`. Alternatively, you could use a virtual machine (you can get one with SQL Server preinstalled!) from Microsoft Azure (`azure.microsoft.com`), particularly if you have an MSDN subscription with the free credit.

To do some of upcoming examples, you will also need the latest `WideWorldImporters` sample database installed for some of the examples, which as of this writing you can find the latest version at `github.com/Microsoft/sql-server-samples/releases/tag/wide-world-importers-v1.0`. I do try my best to maintain proper casing of object names, so if you decide to use a case-sensitive version of the code, it will work.

New for this edition of the book, I will also be including code to do some of the examples using Microsoft's Azure DB offering. You too may want to try Azure DB if this is something that your customers/employers are looking to do.

■ **Tip** I will not be covering much about installation or configuration for the on-premises or cloud versions of SQL Server. The fact is that a simple installation is quite simple, and a complex installation can be very complex indeed. If you need a book on this topic, I would suggest Peter Carter's *Pro SQL Server Administration* (Apress, 2015), a book I helped tech edit, which taught me a lot along the way about how to set up SQL Server in various configurations.

WOW! eBook
www.wowebook.org

CHAPTER 6

Physical Model Implementation Case Study

The whole difference between construction and creation is exactly this: that a thing constructed can only be loved after it is constructed; but a thing created is loved before it exists.

—Charles Dickens, writer and social critic, author of A Christmas Carol

In some respects, the hardest part of the database project is when you actually start to create code. If you really take the time to do the design well, you begin to get attached to the design, largely because you have created something that has not existed before. Once the normalization task is complete, you have pretty much everything ready for implementation, but tasks still need to be performed in the process for completing the transformation from the logical model to the physical, relational model. We are now ready for the finishing touches that will turn the designed 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, we'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 model we have discussed so far in the book is pretty much implementation 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 2016 (or earlier). 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 2016, noting where you would need to adjust if using one of the more recent previous versions of SQL Server, such as SQL Server 2012 or SQL Server 2014.

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_6

WOW! eBook
www.wowebook.org

We will go through the following steps to transform the database from a blueprint into an actual functioning database:

- *Choosing a physical model for your tables*: In the section that describes this step, I will briefly introduce the choice of engine models that are available to your implementation.
- *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.
- *Choosing key implementation*: Throughout the earlier bits of the book, we've made several types of key choices. In the section covering this step, we will go ahead and finalize the implementation keys for the model, discussing the merits of the different implementation methods.
- *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*: The section corresponding to this step provides some basic guidance in creating and naming your schemas. Schema allow you to set up groups of objects that provide groupings for usage and security
- *Adding implementation columns*: We'll consider columns that are common to almost every database that are not part of the logical design.
- *Using Data Definition Language (DDL) to create the database*: In this step's 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, the section for this step offers guidance on how you should approach and implement testing.
- *Deploying your database*: As you complete the DDL and at least some of the testing, you need to create the database for users to use for more than just unit tests. The section covering this step offers a short introduction to the process.

Finally, we'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.

WOW! eBook
www.wowebook.org

■ **Note** For this and subsequent chapters, I'll assume that you have SQL Server 2016 installed on your machine. For the purposes of this book, I recommend you use the Developer edition, which is (as of printing time) available for free as a part of the Visual Studio Dev Essentials from www.visualstudio.com/products/visual-studio-dev-essentials. The Developer Edition gives you all of the functionality of the Enterprise edition of SQL Server for developing software, which is considerable. (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 Azure SQL Database (https://azure.microsoft.com/en-us/services/sql-database/) that I will also make mention of. The features of Azure are constantly being added to, faster than a book can keep up with, but Azure SQL Database will get many features before the box product that I will focus on. I will provide scripts for this chapter that will run on Azure SQL Database with the downloads for this book. Most other examples will run on Azure SQL Database as well.

The main example in this chapter is based on a simple messaging database that a hypothetical company is building for its hypothetical 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).
- 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.

WOW! eBook
www.wowebook.org

Figure 6-1 shows the logical database design for this application, on which I'll base the physical design.

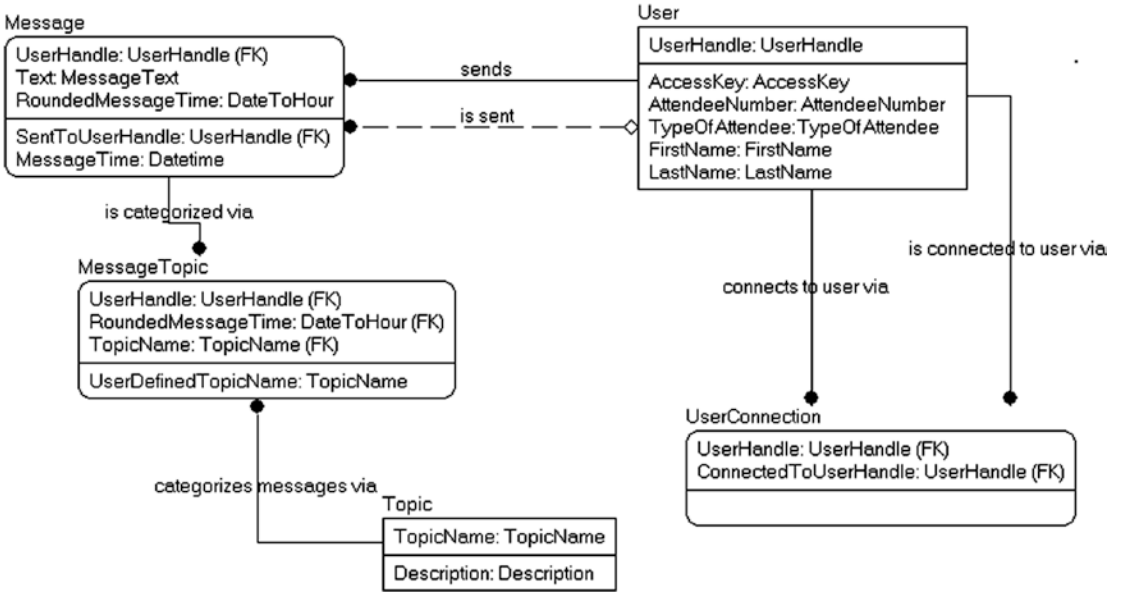


Figure 6-1. *Simple logical model of conferencing message database*

The following is a brief documentation of the tables and columns in the model. To keep things simple, I will expound on the needs as we get to each need individually.

- `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 preloaded with a value based on the person's first and last name, plus an 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.

- `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.

WOW! eBook
www.wowebook.org

- `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 a Physical Model for Your Tables

Prior to SQL Server 2014 there was simply one relational database engine housed in SQL Server. Every table worked the same way (and we liked it, consarn it!). In 2014, a second "*in-memory*" "OLTP" engine (also referred to as "memory optimized") was introduced, which works very differently than the original engine internally but, for SQL programmers, works in basically the same, declarative manner. In this section I will introduce some of the differences at a high level, and will cover the differences in more detail in later chapters. For this chapter, I will also provide a script that will build the tables and code as much as possible using the in-memory engine, to show some of the differences.

Briefly, there are two engines that you can choose for your objects:

- *On-Disk*: The classic model that has been incrementally improved since SQL Server 7.0 was rewritten from the ground up. Data is at rest on disk, but when the query processor uses the data, it is brought into memory first, into pages that mimic the disk structures. Changes are written to memory and the transaction log and then persisted to disk asynchronously. Concurrency/isolation controls are implemented by blocking resources from affecting another connection's resources by using locks (to signal to other processes that you are using a resource, like a table, a row, etc.) and latches (similar to locks, but mostly used for physical resources).
- *In-Memory OLTP*: (Many references will simply be "*in-memory*" for the rest of the book.) Data is always in RAM, in structures that are natively compiled (if you do some just digging into the SQL Server directories, you can find the C code) and compiled at create time. Changes are written to memory and the transaction log, and asynchronously written to a delta file that is used to load memory if you restart

WOW! eBook
www.wowebook.org

the server (there is an option to make the table nondurable; that is, if the server restarts, the data goes away). Concurrency/isolation controls are implemented using versioning (MVCC-Multi-Valued Concurrency Controls), so instead of locking a resource, most concurrency collisions are signaled by an isolation level failure, rather than blocking. The 2014 edition was very limited, with just one unique constraint and no foreign keys, check constraints. The 2016 edition is much improved with support for most needed constraint types.

What is awesome about the engine choice is that you make the setting at a table level, so you can have tables in each model, and those tables can interact in joins, as well as interpreted T-SQL (as opposed to natively compiled T-SQL objects, which will be noted later in this section). The in-memory engine is purpose-built for much higher performance scenarios than the on-disk engine can handle, because it does not use blocking operations for concurrency control. It has two major issues for most common immediate usage:

- All of your data in this model must reside in RAM, and servers with hundreds of gigabytes or terabytes are still quite expensive.
- MVCC is a *very* different concurrency model than many SQL Server applications are built for.

The differences, particularly the latter one, mean that in-memory is not a simple "go faster" button.

Since objects using both engines can reside in the same database, you can take advantage of both as it makes sense. For example, you could have a table of products that uses the on-disk model because it is a very large table with very little write contention, but the table of orders and their line items may need to support tens of thousands of write operations per second. This is just one simple scenario where it may be useful for you to use the in-memory model.

In addition to tables being in-memory, there are stored procedures, functions, and triggers that are natively compiled at create time (using T-SQL DDL, compiled to native code) as well that can reference the in-memory objects (but not on on-disk ones). They are limited in programming language surface, but can certainly be worth it for many scenarios. The implementation of SQL Server 2016 is less limited than that of 2014, but both support far less syntax than normal interpreted T-SQL objects.

There are several scenarios in which you might apply the in-memory model. The Microsoft In-Memory OLTP web page (`msdn.microsoft.com/en-us/library/dn133186.aspx`) recommends it for the following:

- *High data read or insertion rate*: Because there is no structure contention, the In-Memory model can handle far more concurrent operations than the On-Disk model.
- *Intensive processing*: If you are doing a lot of business logic that can be compiled into native code, it will perform far better than the interpreted version.
- *Low latency*: Since all data is contained in RAM and the code can be natively compiled, the amount of execution time can be greatly reduced, particularly when milliseconds (or even microseconds) count for an operation.
- *Scenarios with high-scale load*: Such as session state management, which has very little contention, and may not need to persist for long periods of time (particularly not if the SQL Server machine is restarted).

So when you are plotting out your physical database design, it is useful to consider which mode is needed for your design. In most cases, the on-disk model will be the one that you will want to use (and if you are truly unsure, start with on-disk and adjust from there). It has supported some extremely large data sets in a very efficient manner and will handle an amazing amount of throughput when you design your database properly. However, as memory becomes cheaper, and the engine approaches coding parity with the

WOW! eBook
www.wowebook.org

interpreted model we have had for 20 years, this model may become the default model. For now, databases that need very high throughput (for example, like ticket brokers when the Force awakened) are going to be candidates for employing this model.

If you start out with the on-disk model for your tables, and during testing determine that there are hot spots in your design that could use in-memory, SQL Server provides tools to help you decide if the In-Memory engine is right for your situation, which are describe in the page named "Determining if a Table or Stored Procedure Should Be Ported to In-Memory OLTP" (https://msdn.microsoft.com/en-us/library/dn205133.aspx).

The bottom line at this early part of the implementation chapters of the book is that you should be aware that there are two query processing models available to you as a data architect, and understand the basic differences so throughout the rest of the book when the differences are explained more you understand the basics.

The examples in this chapter will be done solely in the on-disk model, because it would be the most typical place to start, considering the requirements will be for a small user community. If the user community was expanded greatly, the in-memory model would likely be useful, at least for the highest-contention areas such as creating new messages.

■ **Note** Throughout the book there will be small asides to how things may be affected if you were to employ in-memory tables, but the subject is not a major one throughout the book. The topic is covered in greater depth in Chapter 10, "Index Structures and Application," because the internals of how these tables are indexed is important to understand; Chapter 11, "Matters of Concurrency," because concurrency is the biggest difference; and somewhat again in Chapter 13, "Architecting Your System," as we will discuss a bit more about when to choose the engine, and coding using it.

Choosing Names

The target database for our model is (obviously) SQL Server, so our table and column naming conventions must adhere to the rules imposed by this database system, as well as being 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, going back to SQL Server 7.0.

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`. It is defined as a 128-character (or less, of course) string using Unicode characters. SQL Server's rules for identifier consist of two distinct naming methods:

- *Regular identifiers*: Identifiers that need no delimiter, but have a set of rules that govern what can and cannot be a name. 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, as well as 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 ($).

WOW! eBook
www.wowebook.org

    - The name must not be a SQL Server reserved word. You can find a large list of reserved words in SQL Server 2016 Books Online, in the "Reserved Keywords" topic. 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.) Note that some words are considered keywords but not reserved (such as description) and may be used as identifiers. (Some, such as int, would make terrible identifiers!)
    - 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 HR;]` 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 from other environments. Delimiters are generally to be used when scripting objects because a name like `[Drop Database HR;]` 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 (or if you need to consider alternative employment opportunities). 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 to your fellow human 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 in some contexts (see the embedded space) as will `[name]`.

■ **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. (Have you met Darth Vader, development manager? He is nice!)

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 I generally give is: "Whatever you feel is best, as long as others can read it and it follows the local naming standards." This might sound like a cop-out, but there are more naming standards than there are data architects. (On the day this paragraph was first 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 necessary. With space for 128

WOW! eBook
www.wowebook.org

characters, there's little reason to do much abbreviating. A Pascal-cased name is of the form `PartPartPart`, where words are concatenated with nothing separating them. Camel-cased names do not start with a capital letter, such as `partPartPart`.

■ **Caution** Because most companies have existing systems, it's a must to know the shop standard for naming objects 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 rarely pleases anyone.
- `[user connection]` *or* `"user connection"`: This name is delimited by brackets or quotes. Being forced to use delimiters is annoying, and many other languages use double quotes to denote strings. (In SQL, you 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.
- `UserConnection` *or* `userConnection`: Pascal case or camelCase (respectively), using mixed case to delimit between words. I'll use Pascal style in most 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." Some applications that access your data may have limitations like 30 characters that make abbreviations necessary, so understand the needs.

One specific place where abbreviations do make sense are when the abbreviation is *very* standard in the organization. As an example, if you were writing a purchasing system and you were naming 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. Just be 100% certain, so you don't end up with `PO` also representing disgruntled customers along with purchase orders.

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. 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.

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 table is a table, such as `tblSchool` or `tableBuilding`. Please don't do this (really…I beg you). It's clear by the context what is a table. 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

WOW! eBook
www.wowebook.org

context, but this practice is never needed with SQL tables. Note that this dislike of prefixes is just for names that are used by users. We will quietly establish prefixes and naming patterns for non-user-addressable objects as the book continues.

■ **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 `HWWGO1_TAB_USR_CONCT_T` just because the shop standards say so (and yes, I do know when the 19th century was).

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, except for the aforementioned common abbreviations, as well as generally pronounced abbreviations where a value is read naturally as 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.
- Follow a common pattern if possible. For example, a standard that has been attributed as coming from ISO 11179 is to have names constructed in the pattern (RoleName + Attribute + Classword + Scale) where each part is:
  - RoleName [optional]: When you need to explain the purpose of the attribute in the context of the table.
  - Attribute [optional]: The primary purpose of the column being named. If omitted, the name refers to the entity purpose directly.

WOW! eBook
www.wowebook.org

    - Classword: A general suffix that identifies the usage of the column, in non-implementation-specific terms. It should not be the same thing as the datatype. For example, `Id` is a surrogate key, not `IdInt` or `IdGUID`. (If you need to expand or change types but not purpose, it should not affect the name.)
    - Scale [optional]: Tells the user what the scale of the data is when it is not easily discerned, like minutes or seconds; or when the typical currency is dollars and the column represents euros.
- Some example names might be:
    - `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 `decimal(12,2)`, or `money`, or any sort of types).
    - `Pledge`**Amount** Euros is an amount of money in Euros.
    - `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.

Many possible classwords 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. The most important thing is that if you can establish a standard, make it work for your organization and follow it.

■ **Note** Just as with tables, avoid prefixes like col to denote a column as it is a really horrible practice.

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 classwords, like "id" for "identifier"). Later in the book I will use a Hungarian-style prefix for objects other than tables, such as constraints, and for coded objects, such as procedures. This is mostly to keep the names unique and avoid clashes with the table 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 keyword. While `User` is the more natural name than `MessagingUser`, it is one of the trade-offs we have to make 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 if it took me four hours to redraw graphics and

WOW! eBook
www.wowebook.org

undo my original choice of `User` as a table name, I don't want to give you that as a typical practice. If you find you have used a reserved keyword in your model (and you are not writing a chapter in a book that is 80+ pages long about it), it is usually a very minor change.

In the model snippet in Figure 6-2, I have made that change.

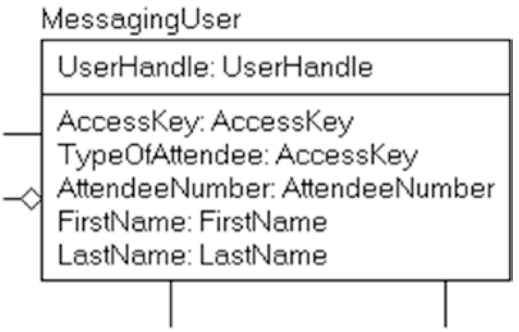


Figure 6-2. *Table User has been changed to MessagingUser*

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 classword at the end of the column name. 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 classword, 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 suffixing `Value` to the name will make the name clearer and distinctive. Figure 6-3 reflects the change in name.

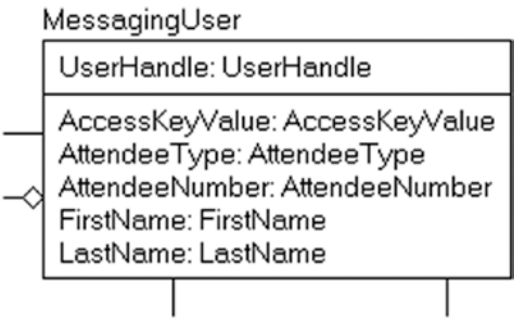


Figure 6-3. *MessagingUser table after change to AccessKey column name*

WOW! eBook
www.wowebook.org

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, 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 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 for all of the tables, of course.

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 for the implementation. 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 used to 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, for example). Then, I'd have an `automobileMake` with a key of `manufacturerName = 'Ford Motor Company'` and `makeName = 'Ford'` (as opposed to Lincoln), `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 value or values that make up a key. Using three 2-byte values could be better than one 15-byte key, though it is a lot more cumbersome to join on three columns.

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 of options, 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.

WOW! eBook
www.wowebook.org

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 artificially generated primary key column and implement alternate keys to protect the uniqueness of the natural keys in your tables, as shown in Figure 6-4.

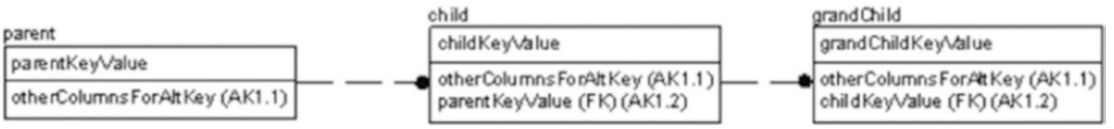


Figure 6-4. *Single-column key example*

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 has not only the primary key but also an alternate key. The term "surrogate" has a very specific meaning, even outside of 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 as small as possible*: If you use numbers, you can use the smallest integer type possible. So if you have a max of 200 rows, you can use a tinyint; 2 billion rows, a 4-byte integer. 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 simple primary keys that will use this index. (Some use a 16-byte GUID for convenience to the UI code, but GUIDs have their downfall, as we will discuss.)
- *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 (usually `Id` in my examples), there will be less thinking to do when setting up the join. Less thinking equates to less errors as well.

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. In our example table in Figure 6-4, you would have to join from the `grandChild` table through the `child` table to get key 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 and foreign keys carefully.

WOW! eBook
www.wowebook.org

Assuming you have chosen to use a surrogate key, the next choice is to decide what values 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 or generated from a `SEQUENCE` object, 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 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 row's surrogate key value (as well as the row) 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, address, etc. 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 generally 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 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.

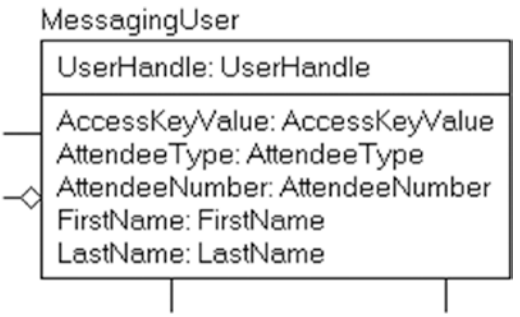


Figure 6-5. *MessagingUser table before changing model to use surrogate key*

WOW! eBook
www.wowebook.org

This value is the most logical, but this name, based on the requirements (current and future), 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 this table and related tables. 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.

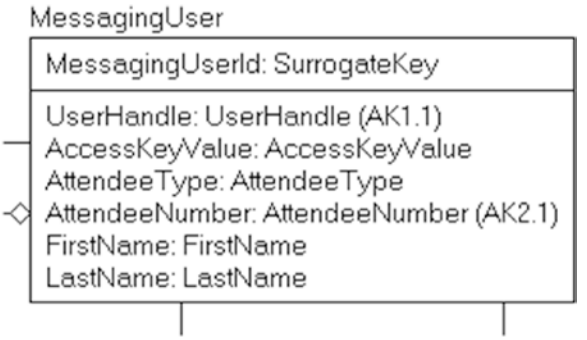


Figure 6-6. *MessagingUser table after changing model to use surrogate key*

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`.

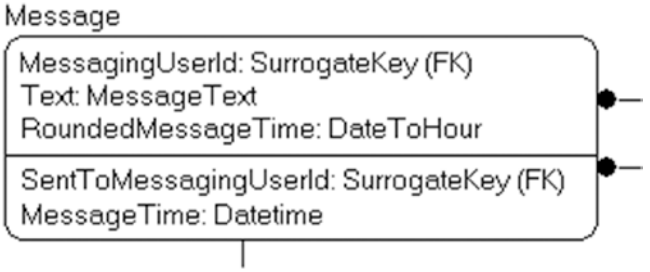


Figure 6-7. *Message table before changing model to use surrogate key*

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.

WOW! eBook
www.wowebook.org

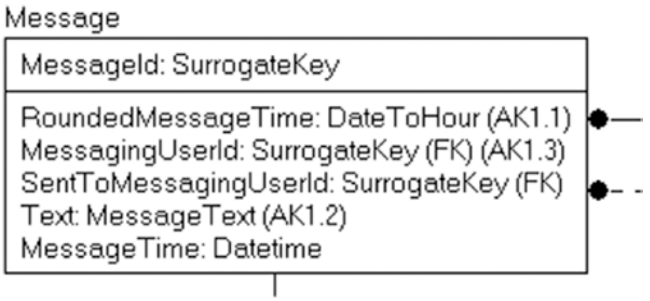


Figure 6-8. *Message table before changing model to use surrogate key*

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 only stands in for 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 secondary copy like a data warehouse. No matter the choice of human-accessible key, surrogate keys are the style of key that I use for nearly all tables in databases I create (and always for tables with user-modifiable data, which I will touch on when we discuss "domain" tables later in this chapter). In Figure 6-9, I have completed the transformation to using surrogate keys.

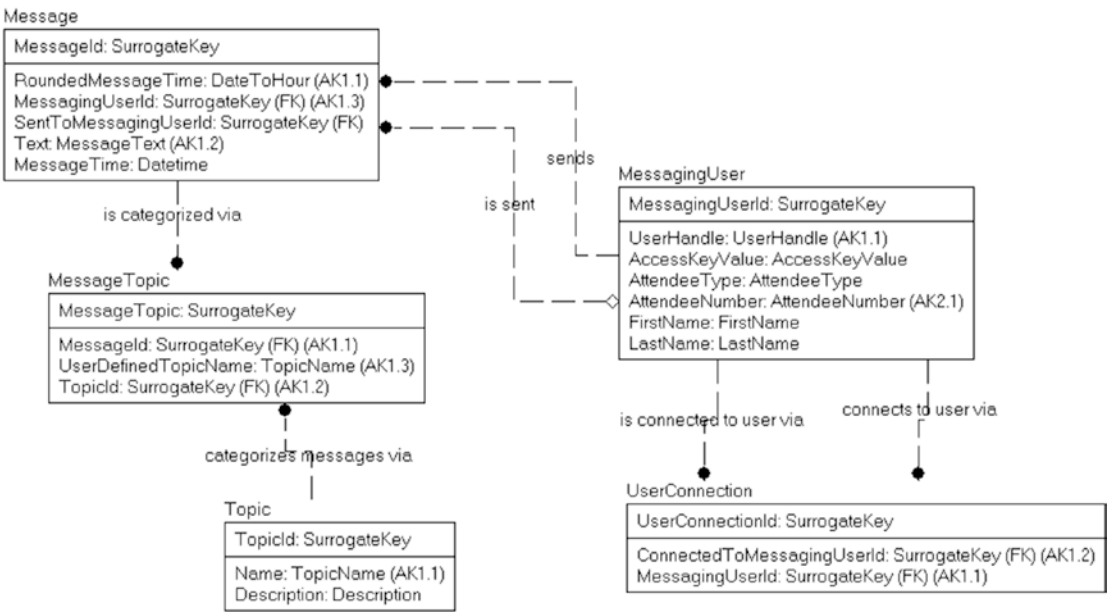


Figure 6-9. *Messaging Database Model progression after surrogate key choices*

WOW! eBook
www.wowebook.org

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. With only an artificially generated value as your key, 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 that could occur in the same .000001 of a second), 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.

WOW! eBook
www.wowebook.org

In a well-designed logical model, you should not have anything to do at this point with keys that protect the uniqueness of the data from a requirements standpoint. 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`.

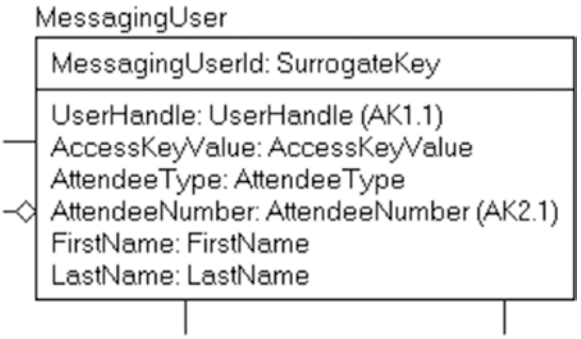


Figure 6-10. *MessagingUser table for review*

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`.

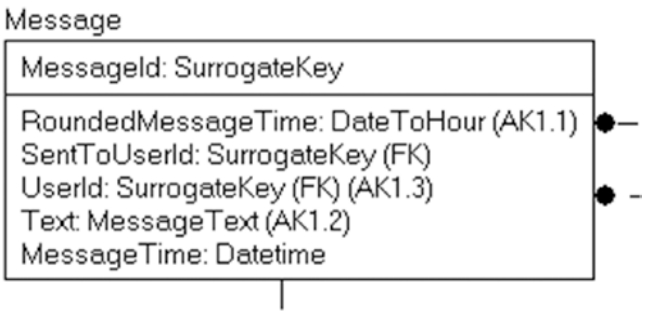


Figure 6-11. *Message table for review*

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 (which we will find some way to implement later in the process), configuring the structures 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, typically by translating the error message to something nicer.

WOW! eBook
www.wowebook.org

The last table I will cover here is the `MessageTopic` table, shown in Figure 6-12.

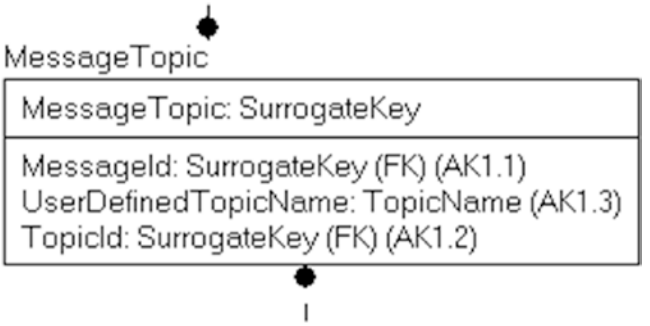


Figure 6-12. *MessageTopic table for review*

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.

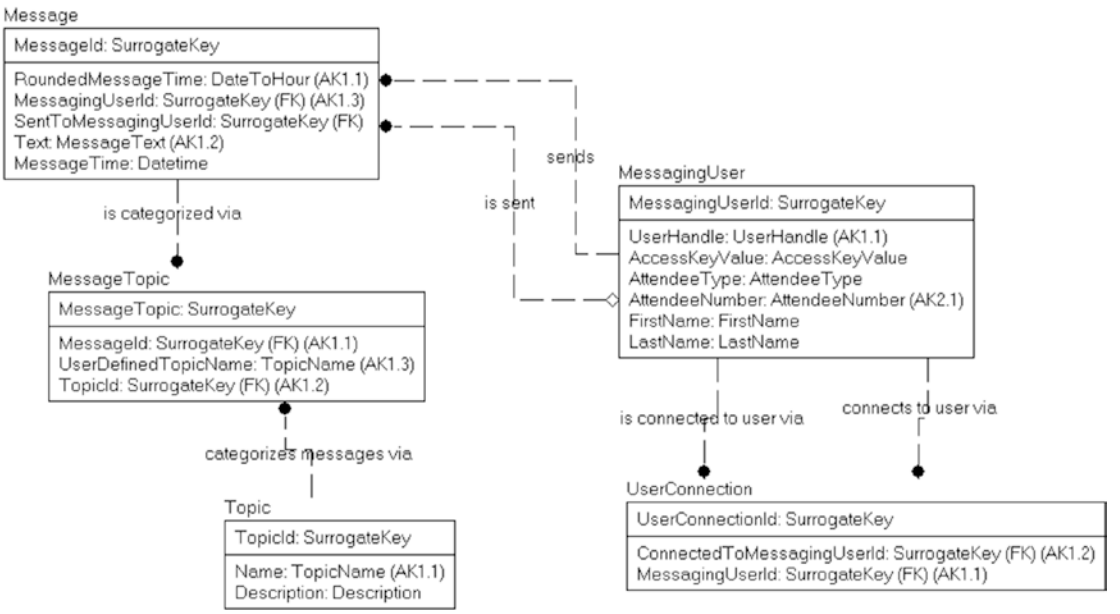


Figure 6-13. *Messaging model for review*

WOW! eBook
www.wowebook.org

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 (it can seem like work to be a pompous data architect, rather than a programmer), but it becomes clear during physical modeling if it has been done well up front. During implementation domains serve several purposes:

- *Consistency*: We have used `TopicName` twice as a domain. This reminds us to define every column of type `TopicName` 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! If you have tool support for property inheritance on domains, when you change a property in the definition, the values change everywhere. So if all descriptive-type columns are nullable, and all code columns are not, you can set this in one place.
- *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 of the implemented columns.

Domains aren't a requirement of logical or physical database design, nor does SQL Server actually make it easy for you to use 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`

WOW! eBook
www.wowebook.org

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 usages later in this chapter.

A second domain that is used very often in our model 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 in the relationship.
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, if using identity columns for the surrogate, 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 will not allow them. Finally, let's set up one more domain definition to our sample, the UserHandle domain, shown in Table 6-4.

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 several topics concerning the implementation of 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 this section, I will demonstrate how to implement the datatype choices in the example model.
- *Choosing the collation*: The collation determines how data is sorted and compared, based on character set and language used.

WOW! eBook
www.wowebook.org

Getting the domain of a column implemented correctly is an important step in getting the implementation correct. Too many databases end up with all columns with the same datatype and size, allowing nulls (except for primary keys, if they have them), and lose the integrity of having properly sized and constrained constraints.

Enforce Domain in the Column, or With a 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`.

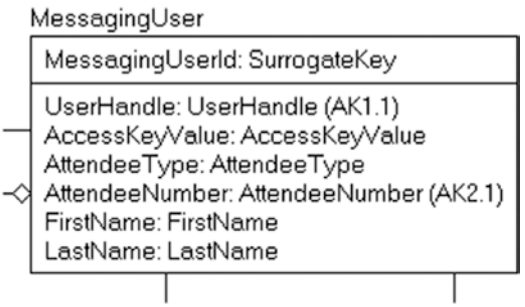


Figure 6-14. *MessageUser table for reference*

This domain might be specified as in Table 6-5.

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

WOW! eBook
www.wowebook.org

The value limitation limits the values to a fixed list of values. We could choose to implement the column using a declarative control (a `CHECK` constraint, which we will cover in more detail later in the chapter) 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 and you do a `DISTINCT` query over the column (something that is generally poorly performing), 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 attendee types 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. 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 for a domain table can be a bit different than for most tables. Sometimes, I use a surrogate key for the actual primary key, and other times, I use a natural key. The general difference is whether or not the values are user manageable, and if the programming tools require the integer/GUID approach (for example, if the front-end code uses an enumeration that is being reflected in the table values). 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.

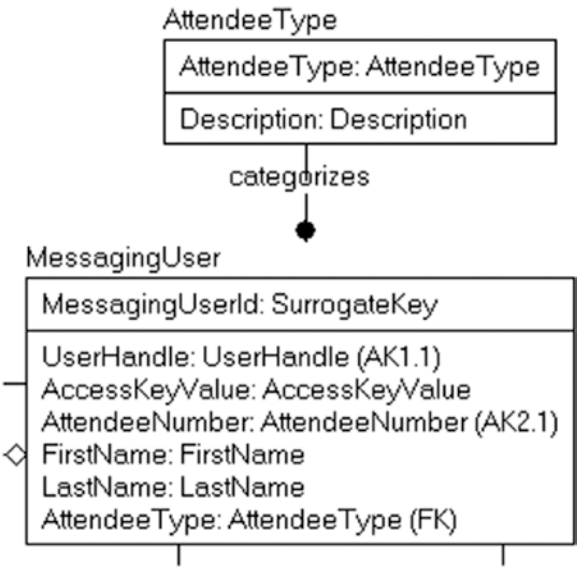


Figure 6-15. *AttendeeType domain implemented as a table*

WOW! eBook
www.wowebook.org

This lets an application treat the value as if it is a simple value just like if this was implemented without the domain table. So if the application wants to manage the value as a simple string value, 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, plus the ability to have a `Description` column describing what each of the values actually means (which really comes in handy at 12:10 AM on December the 25th when the system is crashing and needs to be fixed, all while you are really thinking about the bicycle you haven't finished putting together).

In the original model, we had the `Topic` table, shown in Figure 6-16, which is a domain similar to the `AttendeeType` but is designed to allow a user to make changes to the topic list.

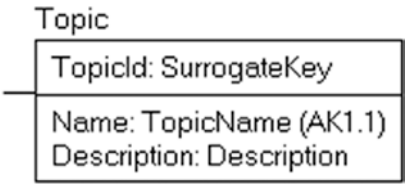


Figure 6-16. *Topic table for reference*

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 user-defined topic that allows the user to enter their own topic in the `MessageTopic` table.

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 (built-in types that are installed when you install SQL Server) and a brief explanation of each of them. As you are translating domains to implementation, step 1 will be to see which of these types matches to need best first, then we will look to constrain the data even further with additional techniques.

- *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. Some typical integer operations, like basic math, cannot be performed.
  - `tinyint`: Non-negative values between `0` and `255` (0 to 2^8 - 1) (1 byte).

WOW! eBook
www.wowebook.org

    - `smallint`: Integers between `-32,768` and `32,767` (-2^15 to 2^15 – 1) (2 bytes).
    - `int`: Integers between -2,147,483,648 and 2,147,483,647 (–2^31 to 2^31 – 1) (4 bytes).
    - `bigint`: Integers between 9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 (-2^63 to 2^63 – 1) (8 bytes).
    - `decimal` *(or* `numeric`*, which is functionally the same in SQL Server, but decimal is generally preferred for portability)*: All numbers between –10^38 – 1 and 10^38 – 1 (between 5 and 17 bytes, depending on precision). Allows for fractional numbers, unlike integer-suffixed types.

- *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 an ISO synonym for a `float(24)` datatype, and hence equivalent (4 bytes).

- *Date and time*: Stores values that deal with temporal data.
    - `date`: Date-only values from January 1, 0001, to December 31, 9999 (3 bytes).
    - `time`: Time-of-day-only values to 100 nanoseconds (3 to 5 bytes).
    - `datetime2(N)`: Despite the hideous name, this type will store a point in time from January 1, 0001, to December 31, 9999, with accuracy ranging from 1 second (0) to 100 nanoseconds (7) (6 to 8 bytes).
    - `datetimeoffset`: Same as `datetime2`, but includes an offset for time zone (8 to 10 bytes).
    - `smalldatetime`: A point in time 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`: Points in time 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 (2GB) long.

WOW! eBook
www.wowebook.org

- *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 (2GB) long.
  - `nchar`, `nvarchar`, `nvarchar(max)`: Unicode equivalents of `char`, `varchar`, and `varchar(max)`.
- Other datatypes:
  - `sql_variant`: Stores (pretty much) any datatype, other than CLR-based datatypes (`hierarchyId`, spatial types) and any types with a max length of over 8,016 bytes. (CLR is a topic I won't hit on too much, but it allows Microsoft and you to program SQL Server objects in a .NET language. For more information, check `https://msdn.microsoft.com/en-us/library/ms131089.aspx`). It's generally a bad idea to use `sql_variant` for all but a few fringe uses. It is usable in cases where you don't know the datatype of a value before storing. The only use of this type in the book will be in Chapter 8 when we create user-extensible schemas (which is itself a fringe pattern.)
  - `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 make further use of the `rowversion` datatype in more detail in Chapter 11, which is about concurrency. (16 years later, and it is still referred to as timestamp very often, so this may never actually go away completely.)
  - `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 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 these types in this book.
  - `hierarchyId`: 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

WOW! eBook
www.wowebook.org

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 (or even the newer WideWorldImporters database, which is less unrealistically complex than AdventureWorks and will be used in several chapters later in the book).

Deprecated or Bad Choice Types

I didn't include several datatypes in the previous list because they have been deprecated for quite some time, and it wouldn't be surprising if they are completely removed from the version of SQL Server after 2016 (even though I said the same thing in the previous few versions 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)

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 it 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 input allows for 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 the appendix I will include a more thorough example of the types of issues you could 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

Boolean values (TRUE or FALSE) 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.

WOW! eBook
www.wowebook.org

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 the .NET languages with no translation. The check box and option controls can directly connect to these values, even though a language like VB used `-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 classword, 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.
- *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 two values, since the database structure needn't change if you need 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, why they did it, when it took place, and perhaps even when it takes effect (these are all questions for design time, but it doesn't hurt to be thorough).

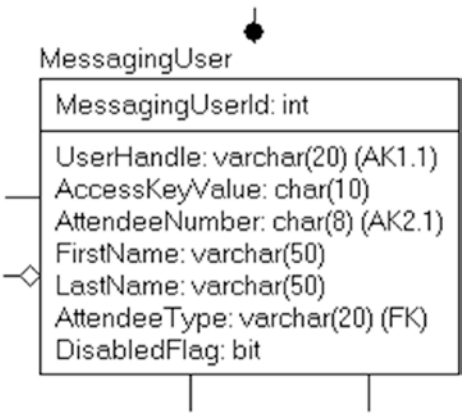


Figure 6-17. *MessagingUser table with DisabledFlag bit column*

WOW! eBook
www.wowebook.org

Large-Value Datatype Columns

In SQL Server 2005, dealing with large datatypes changed quite a bit (and hopefully someday Microsoft will kill the `text` and `image` types for good). 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, 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, as you start to implement. Most databases have a "comments" column somewhere that morphed from comments to a semistructured mess that your DBA staff then needs to dissect using `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 to allow partial column modifications.

The first issue is pretty simple, but it can be a bit confusing at times. For example, concatenate `'12345' + '67890'`. You've taken two `char(5)` values, and the result will be contained in a value that is automatically recast as a `char(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 8000, as the two columns concatenate to a type of `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 2GB, 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 2GB of physical

WOW! eBook
www.wowebook.org

RAM, so paging would likely occur on the client machine, and the whole process would crawl and more than likely crash occasionally. 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 be quite large. In brief, a row that fits into 8,060 bytes can be stored in one physical unit. If the row is larger than 8,060 bytes, string data can be placed on what are called *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. (Physical structures, including 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 huge 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. Versions of SQL Server starting in 2008 allow 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 file tables.

User-Defined Type/Alias

One really excellent sounding feature that you can use to help make your code cleaner is a user-defined type (UDT), which is really an alias to a type. You can use a 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];
```

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 an alias type (variables will always be nullable). In general, it is best to always specify the nullability in the table declaration, and something I will do always in the book, though I do sometimes forget in real life.

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 `CREATE TABLE` statement, we could specify

```
CREATE TABLE MessagingUser
...
UserHandle UserHandle,
```


WOW! eBook
www.wowebook.org

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 making a change to the type a two step process.

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 better, but I do want to make sure that you have at least heard of them and know the pros and cons.

Complex CLR Datatypes

In SQL Server 2005 and later, we can build our own datatypes using the SQL CLR (Common Language Runtime). 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.

For the most part you should use CLR types 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 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 complex business problem.

Microsoft has provided several intrinsic types based on the CLR 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 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 more or less an array of connected points.

Choosing the Right Datatype

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 in which every single column is either an integer or a varchar(N) (where N is the same for every single string column, sometimes in the 8000 range) 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 can be stored as a 16-byte binary value, whereas if you use a varchar column, it will take 36 bytes;

WOW! eBook
www.wowebook.org

however, store it in an `nvarchar` (Unicode) column, and now it takes 72 bytes! What a terrible waste of space. Even worse, someone could put in a non-GUID value up to 200 characters wide. 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 at all. The name should be in something such as a `nvarchar(50)` column and the date of birth in a `date` column. Notice that I used a variable-sized 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 50 bytes, using a variable-sized type will save space in your database. I used a Unicode type because person's names do actually fit the need of allowing nontypical Latin characters.

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 to keep in mind that the datatype is the first level of domain enforcement. Thinking back to our domain for `UserHandle`, we had the datatype definition and value limitations specified in Table 6-6.

Table 6-6. *Sample Domain:UserHandle*

Property	Setting
Name	UserHandle
Optional	`no`
Datatype	Basic character set, maximum 20 characters
Value Limitations	Must be 5-20 simple alphanumeric characters 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 7 I'll discuss patterns of integrity enforcement that are more complex. In this case, I do use a simple ASCII character set because the requirements called for simple alphanumeric data.

Initially, we had the model in Figure 6-18 for the `MessagingUser` table.

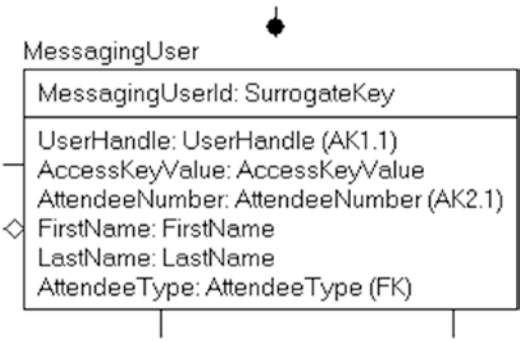


Figure 6-18. *MessagingUser table before choosing exact datatypes*

WOW! eBook
www.wowebook.org

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.`

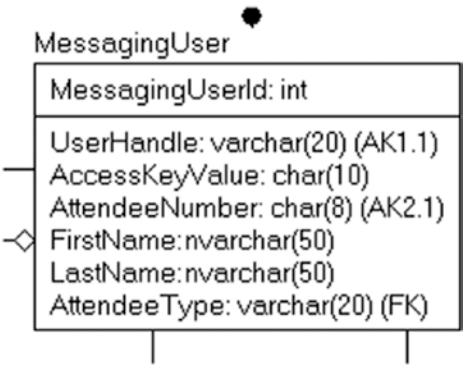


Figure 6-19. *MessagingUser after datatype choice*

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.

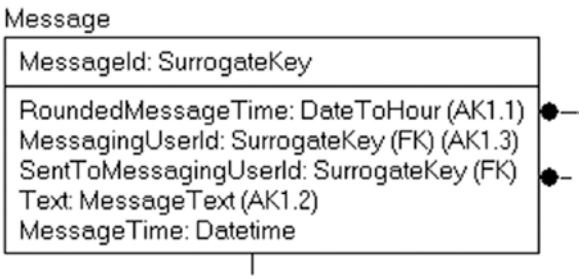


Figure 6-20. *Message table before datatype choice*

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 will be 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.

WOW! eBook
www.wowebook.org

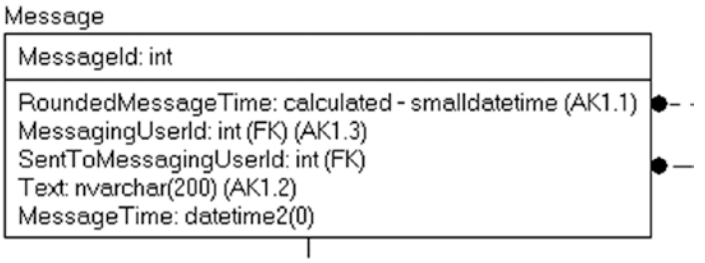


Figure 6-21. *Message table after datatype choice, with calculated column denoted*

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.

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.

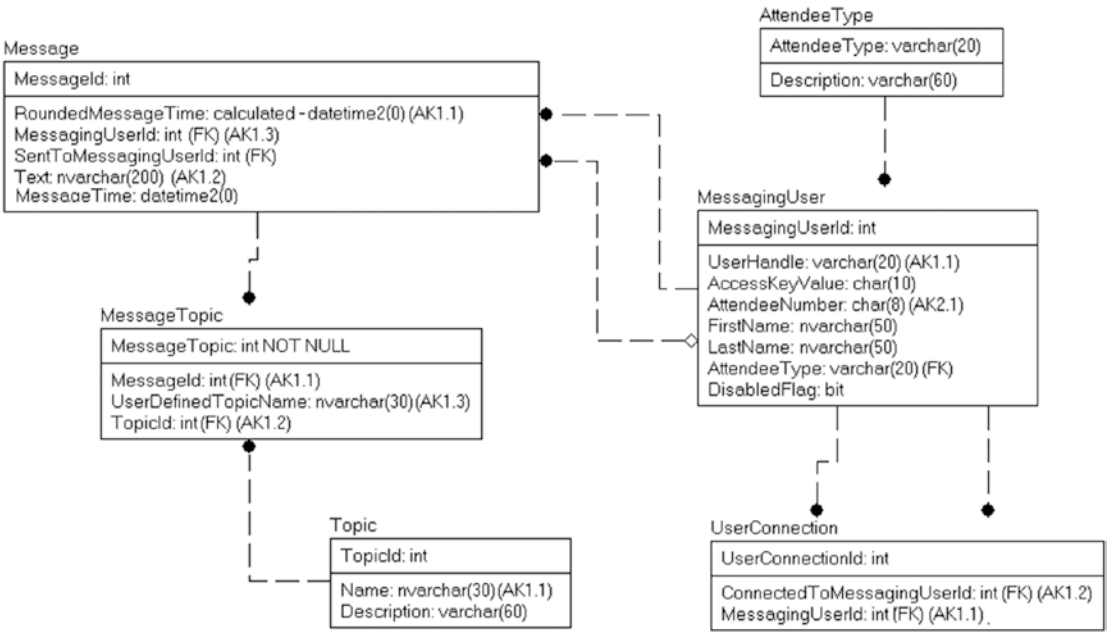


Figure 6-22. *Messaging system model after datatype choices*

Setting 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 nullability settings for the columns.

WOW! eBook
www.wowebook.org

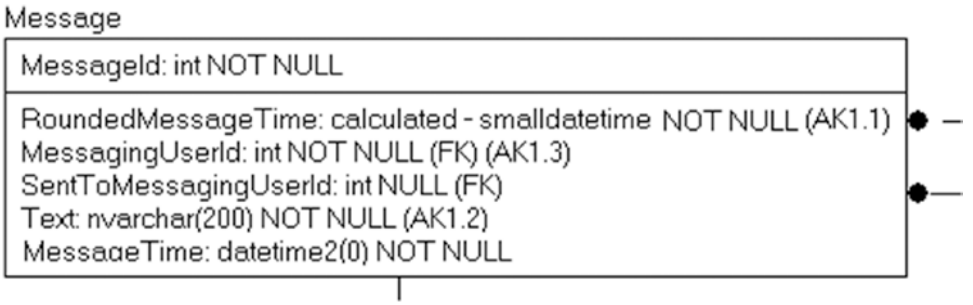


Figure 6-23. *Message table for review*

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.

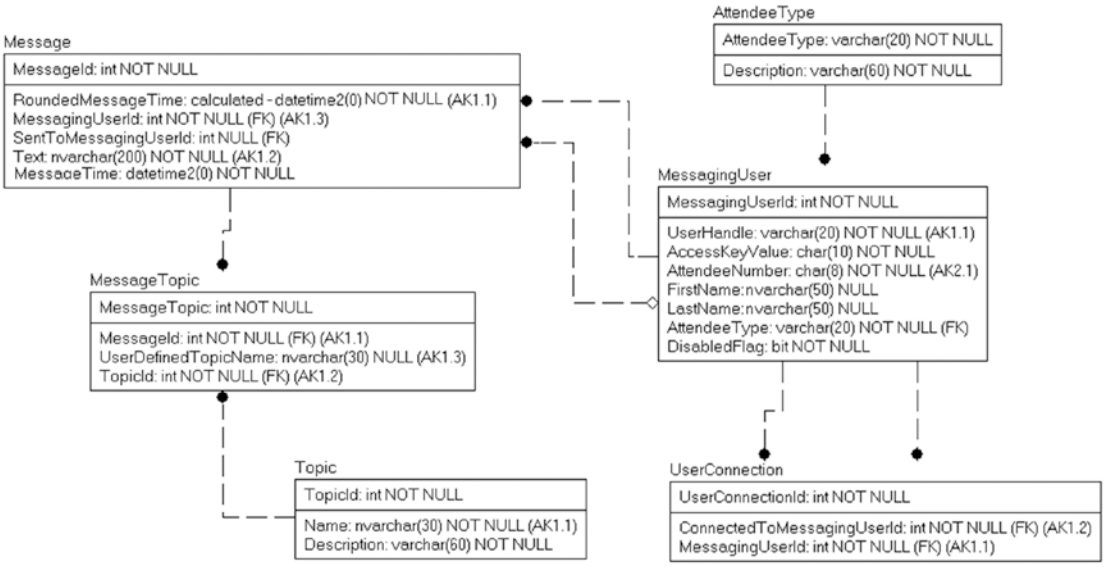


Figure 6-24. *Messaging system model, with NULLs chosen*

At this point, our model is very nearly done and very much resembles a database that could be built and employed in an application. Just a bit more information is needed to finish out the model.

Choosing the Collation

The collation of a string value sets up the way a string is compared with another string, as well as how they are sorted. 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). SQL Server and

WOW! eBook
www.wowebook.org

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 typical for all users of a system. 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('DatabaseName','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 documentation. However, the default is rarely the desired collation to use, and is set to that older collation for backward compatibility. Ideally, you will use a Windows collation, and I will use `Latin1_General_100_CI_AS` for my example code, which will be what I have installed my server as. The `100` indicates this is a newer collation that supports later Unicode characters.

In addition to the *normal* collations, there are also binary collations that you can use to sort and compare data based on in its raw format. There are two types of binary collations: the older ones are suffixed _bin, and newer ones _bin2. Bin2 collations do what is referred to as pure code-point collations, meaning they compare data only on the binary value (the bin collation compared the first byte as a WCHAR, which is an OLEDB datatype). Note that there will be a difference between binary sort order and case sensitive, so it will behoove you to take some time to understand the collation that you end up using.

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 3,800 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 SCHEMA alt;
CREATE TABLE alt.OtherCollate
(
   OtherCollateId int 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
);
```


WOW! eBook
www.wowebook.org

Now, when you sort output by FrenchName, it's case insensitive, but arranges the rows according to the order of the French language. The same applies with Spanish, regarding the SpanishName column. For this chapter we will stick with the default in almost all cases, and I would suggest taking a look at Books Online if you have the need to store data in multiple languages. The only "normal" variation from the default collation of the database is if you need a different case sensitivity for a column, in which case you might use a binary collation, or a case sensitive one.

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
```

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.

If you do find yourself on a server with multiple collations, a handy collation setting to use can be database_default, which uses the default for the context of the database you are executing from.

It is important to choose your collation wisely when setting up a server, database, table, etc. Changing the collation of a database can be done using a simple ALTER statement. However, it will not change any objects in the database. You will have to change every column's collation individually, and while you can use a simple ALTER TABLE…ALTER COLUMN statement to change the collation, you will have to drop all indexes, constraints, and schema-bound objects that reference the column first. It is a pretty painful task.

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 schemas to group our tables and eventually views, procedures, functions, etc. into functional groups. Naming schemas is a bit different than naming 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.

WOW! eBook
www.wowebook.org

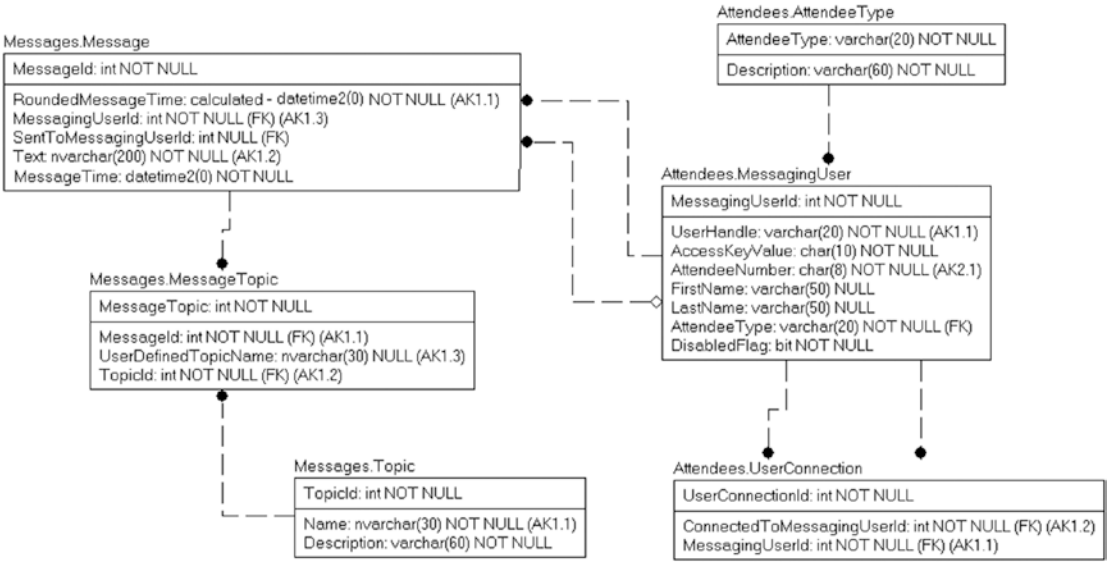


Figure 6-25. *Messages model with schemas assigned*

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.

I'm not going to go any further into the security aspects of using schemas at this point in the book, but understand that schemas are not just for aesthetics. 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. On a brief trip back to the land of reality, I said in the previous editions of this book that beginning to use schemas in production systems will be a slow process, and it still can be jarring to some users 11+ years later. Chapter 9 will discuss using schemas for security in more detail.

Adding Implementation Columns

Finally, I will add one more thing to the database: columns to support the implementation of the code only (and not to support a user requirement directly). A very common use is to have columns that indicate when the row was created, when it was updated, and perhaps by whom. 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 I have one clock managing times, rather than multiples. (I once managed a system that used two clocks to set row times, and occasionally a row was created years after it was last updated!)

So in Figure 6-26, I add two `NOT NULL` columns to every table for the `RowCreateTime` and `RowLastUpdateTime`, except for the `AttendeeType` table, which we specified to be not user manageable, and so I chose not to include the modified columns for that table. Of course, you might want to do this to let your development team know when the row was first available. I also left off columns to denote who changed the row for simplicity.

WOW! eBook
www.wowebook.org

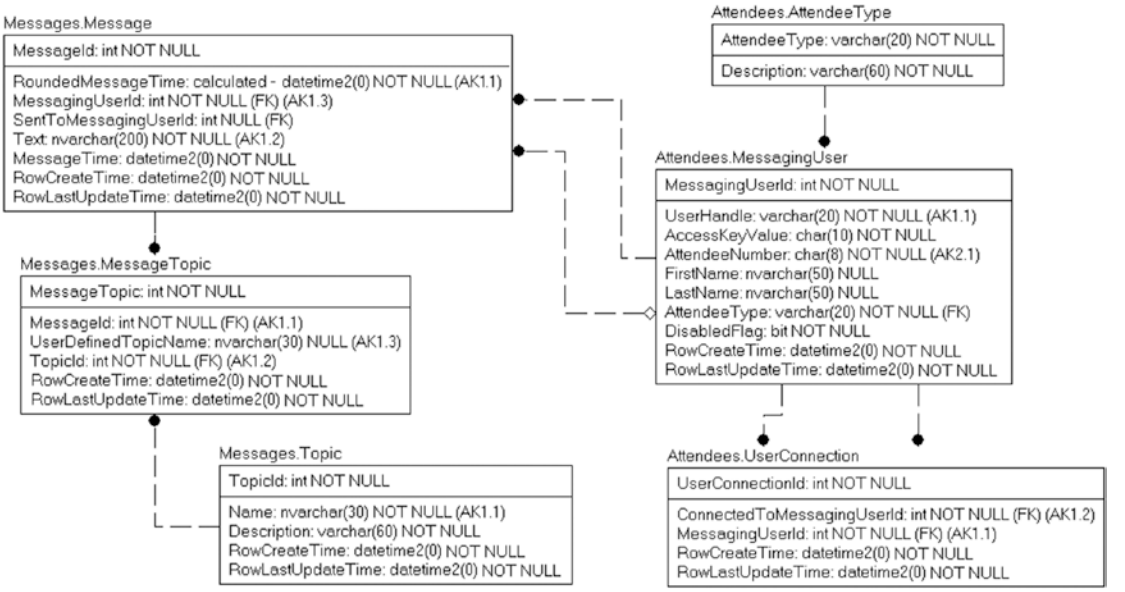


Figure 6-26. *Message model after adding RowCreateTime and RowLastUpdateTime to tables*

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, captured from a different clock source, even if it took five minutes to actually store the data. Plus, if you need to load the data into a new table, the row may have been created in 2016, but the data in 2000, so not using these columns as user data means you can truthfully reflect when the row was created.

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 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 to denote when a row was changed, but when I have control over the design, 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 Chapter 11.

Using DDL to Create the Database

So far, we have been molding the model to make it fit our needs to implement. We added columns, added tables, and specified constraints. 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 building a database with 300 tables without a data modeling tool, but I definitely do know people who do 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

WOW! eBook
www.wowebook.org

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.

Make sure that your scripts are in a source control system too, or at the very least backed up. In SQL Server, we have two tools that we can work in, Management Studio and Visual Studio Data Tools. Data Tools is the development-oriented tool that allows a developer to work in a manner kind of like a .Net developer would. Management Studio is more administrative oriented in its toolset, but has tools to view and edit objects directly.

In this book, I am going to stick to the DDL that any of the tools will use to construct objects using an online paradigm where I create a database directly one command at a time. Such scripts can be executed using any SQL Server tool, but I will generally just use a query window in Management Studio, or sometimes the SQLCMD.exe command-line tools when executing multiple scripts. You can download Management Studio or Data Tools from the Microsoft web site for free at `msdn.microsoft.com/library/mt238290.aspx` or `msdn.microsoft.com/en-us/library/mt204009.aspx`, respectively, though these locations are certainly apt to change in the years after this book is released.

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 a Hyper-V VM on my laptop (most any VM technology will do, or you can install on most modern versions of Windows; and as of this writing, a version of SQL Server running on Linux is being previewed, and the crust of the earth has not frozen over). I use the Developer Edition, and I used most of the default settings when installing (other than setting up mixed mode for security to allow for some security testing later, and the collation, which I will set to `Latin1_General_100_CI_AS`), which will not be optimal when setting up a server for real multiuser use, or if you want to test features like Always On, Replication, etc. All users of a server for the material I am presenting can be managed and created in SQL Server code alone.

If you are using a shared server, such as a corporate development server, you'll need to do this with an account that has rights to create a database. If you install your server yourself, part of the process will be to set up users so the server will be accessible.

Choosing a database name is in the same level of importance 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.
- *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.

WOW! eBook
www.wowebook.org

I will use the following statement to create a small database: `CREATE DATABASE ConferenceMessaging;` You can see where the database files were placed by running the following statement (note that size is presented in 8KB 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         8           C:\Program Files\Microsoft...SQL\DATA\ConferenceMessaging.mdf
LOG          8           C:\Program Files\Microsoft...SQL\DATA\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:

```
USE ConferenceMessaging;

--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 `louis` with a machine named `WIN-8F59BO5AP7D`:

```
databaseOwner
--------------------
WIN-8F59BO5AP7D\louis
```

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 users. 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 9). You can change the owner of the database by using the `ALTER AUTHORIZATION` statement:

```
ALTER AUTHORIZATION ON DATABASE::ConferenceMessaging TO SA;
```

Going back and using the code to see the database owner, you will see that the owner is now `SA`.

WOW! eBook
www.wowebook.org

■ **Tip** Placing a semicolon at the end of every statement in your T-SQL is fast becoming a standard that will, in a future version 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 use either an on-disk or in-memory configuration, place the table on a filegroup, partition the table onto multiple filegroups, control where maximum/overflow data is placed, and so on. Some of this will be discussed in Chapter 10 on table structures and indexing. As will be the typical normal for most databases created, we will be using on-disk tables for most of the examples in the book.

■ **Tip** Don't make this your only source of information about DDL in SQL Server. Books Online is another great place to get exhaustive coverage of 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. Many of the remaining chapters of the book will delve into the more complex usage patterns, but even then we will not cover every possible, even useful, setting that exists.

The base `CREATE` clause is straightforward:

```
CREATE TABLE [<database>.][<schema>.]<tablename>
```

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 only be able to create objects 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. By default the schema will be dbo (or you can set a default schema by database principal…it is far better to specify the schema at all times).
- `<tablename>`: This is the name of the table.

WOW! eBook
www.wowebook.org

For the table name, if the first character 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 they don't really pertain to the database design. You can also declare a local variable table that has the same scope as a variable by using an @ in front of the name, which can be used to hold small sets of data.

The combination of `schema` and `tablename` must be unique in a database, and `tablename` must be unique from any other objects in the database, including include tables, views, procedures, constraints, and functions, among other things. It is why I will suggest a prefix or naming pattern for objects other than tables. Some things that look like objects are not, such as indexes. This will all be clearer as we progress through the book.

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 objects within the schema.

In SQL Server 2000 and earlier, the table was owned by a user, which made using schemas 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. You can access objects using the following naming method, just like when using the `CREATE TABLE` statement:

```
[<databaseName>.][<schemaName>.]objectName
```

The `<databaseName>` defaults to the current database. The `<schemaName>` defaults to the user's default schema.

Schemas are of great use to segregate objects within a database for clarity of use. In our database, we have already specified two schemas earlier: `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
```

The `CREATE SCHEMA` statement has another variation where you create the objects that are contained within it that is rarely used, as in the following example:

```
CREATE SCHEMA Example
      CREATE TABLE ExampleTableName ... ; --no schema name on the object
```

The schema will be created, as well as any objects you include in the script, which will be members of that schema. If you need to drop a schema (like if you created the `Example` schema to try out the syntax like I did!), use `DROP SCHEMA Example;`.

You can view the schemas that have been 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
```


WOW! eBook
www.wowebook.org

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. Or sometimes a user will get a schema with the same name as their user built by some of the SQL Server tools (which is the purpose of WHERE name <> USER_NAME(principal_id) in the schema query). You can change the ownership using the ALTER AUTHORIZATION statement much 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;
```

The ALTER USER command allows the changing of default schema for existing users (and in SQL Server 2012 and later, it 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.
There are two types of columns:

- *Implemented*: This is an ordinary column, in which physical storage is always 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. These may be stored or calculated only when accessed.

WOW! eBook
www.wowebook.org

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.

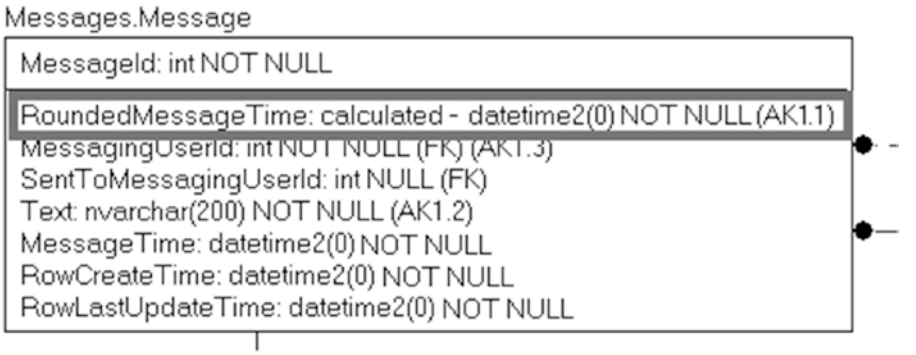


Figure 6-27. *Message table with computed column highlighted*

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 an expression 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. 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 @pointInTime datetime2(0);
set @pointInTime = SYSDATETIME();
```

Next, I write the following expression:

```
DATEADD(HOUR,DATEPART(HOUR,@pointInTime),CAST(CAST(@PointInTime 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 the expression is tested, you replace the variable with the `MessageTime` column and we define our calculated column as follows:

```
,RoundedMessageTime AS DATEADD(HOUR,DATEPART(HOUR,MessageTime),
                        CAST(CAST(MessageTime AS date) AS datetime2(0)) ) PERSISTED
```


WOW! eBook
www.wowebook.org

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, meaning that using the column will not require the value to be calculated at runtime. 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 back in Chapter 5). You can also use a computed column based on 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 to use in a computed column, but you could not persist or index it, since the value would change for every execution.

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, which is governed by the ANSI_NULL_DFLT_OFF and ANSI_NULL_DFLT_ON database properties (see https://msdn.microsoft.com/en-us/library/ms187356.aspx for more details). It is very much a best practice to always specify the nullability of a column, but I won't attempt to demonstrate how those settings work, as they are fairly confusing.

Managing Non-natural 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 colors, you might use 'BL' for Blue, and 'GR' for Green.
- Automatically generated, using the IDENTITY property or a GUID stored in a uniqueidentifier type column.

WOW! eBook
www.wowebook.org

- Automatically generated, but using a DEFAULT constraint, allowing you to override values if it is desirable. For this you might use a GUID using the NEWID() or NEWSEQUENTIALID() functions, or a value generated based on a SEQUENCE object's values.

Of course, if your tables don't use any sort of surrogate values, you can move on to the next sections.

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 and allows usage of the 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).

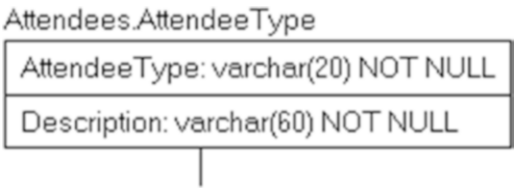


Figure 6-28. *AttendeeType table for reference*

Note that it's generally expected that once you manually create a value for a table that is build to implement a domain, 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 privileges'). 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 can be 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 as an autonomous transaction that is outside of the normal transaction so it works extremely fast and doesn't lock other connections from anything other than the generation of a new sequential value. 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 ,
```


WOW! eBook
www.wowebook.org

```
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 new rows, 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 manually. 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 are able to 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, there are a couple of things you can use in a `DEFAULT` constraint to set values when a value isn't provided.

First, if using GUIDs for a key, you can simply default the column to `NEWID()`, which will generate a random GUID value. Or you can use `NEWSEQUENTIALID()`, which will generate a GUID with a higher value each time it is called. Having a monotonically increasing sequence of values has benefits to indexes, something we will discuss more in Chapter 10.

Another method of generating an integer value for a surrogate is to use a `SEQUENCE` object to generate new values for you. Like an identity column, it is not subject to the primary transaction, 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. The only real concern with using a `SEQUENCE` based surrogate key is that you

WOW! eBook
www.wowebook.org

are not limited to using the values the SEQUENCE object generates. So if someone enters a value of 10 in the column without getting it from the SEQUENCE object, you may get a key violation. SEQUENCE objects have techniques to let you allocate (or sometimes referred to as *burn*) sets of data (which I will cover in a few pages).

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
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 two values using the NEXT VALUE statement for sequence objects:

```
SELECT NEXT VALUE FOR Messages.TopicIdGenerator AS TopicId;
SELECT NEXT VALUE FOR Messages.TopicIdGenerator AS TopicId;
```

This returns

```
TopicId
-----------
10000

TopicId
-----------
10001
```

■ **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;
```


WOW! eBook
www.wowebook.org

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 use a default, so I will note that the name I gave the default object starts with a prefix of DFLT, followed by the 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 DFLTTopic_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 how 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;

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 (which 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 SEQUENCE objects in the database, you can use the sys.sequences catalog view:

```
SELECT start_value, increment, current_value
FROM sys.sequences
WHERE SCHEMA_NAME(schema_id) = 'Messages'
   AND name = 'TopicIdGenerator';
```

For the TopicGenerator object we set up, this returns

```
start_value     increment      current_value
--------------- -------------- --------------------
10000           1              10099
```


WOW! eBook
www.wowebook.org

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`.

The Actual DDL to Build Tables

We have finally reached the point where we are going to create the basic table structures we have specified, including generating the primary keys and the calculated column that we created. Note that we have already created the `SCHEMA` and `SEQUENCE` objects earlier in the chapter. I will start the script with a statement to drop the objects if they already exist, as this lets you create and try out the code in a testing manner. At the end of the chapter I will discuss strategies for versioning your code, keeping a clean database from a script, either by dropping the objects or by dropping and re-creating a database.

```
--DROP TABLE IF EXISTS is new in SQL Server 2016. The download will have a method for older
--versions of SQL Server demonstrated as well. The order of the tables is actually set based
--on the order they will need to be dropped due to later foreign key constraint order
DROP TABLE IF EXISTS Attendees.UserConnection,
                     Messages.MessageTopic,
                     Messages.Topic,
                     Messages.Message,
                     Attendees.AttendeeType,
                     Attendees.MessagingUser
```

Now we will create all of the objects:

```
CREATE TABLE Attendees.AttendeeType (
        AttendeeType         varchar(20)  NOT NULL ,
        Description          varchar(60)  NOT NULL
);
--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            nvarchar(50)  NULL ,
        LastName             nvarchar(50)  NULL ,
        AttendeeType         varchar(20)  NOT NULL ,
        DisabledFlag         bit  NOT NULL ,
        RowCreateTime        datetime2(0)  NOT NULL ,
        RowLastUpdateTime    datetime2(0)  NOT NULL
);
```


WOW! eBook
www.wowebook.org

```
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
);

CREATE TABLE Messages.Topic (
        TopicId int NOT NULL CONSTRAINT DFLTTopic_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.

■ **Note** If you are trying to create a table (or any object, setting, security principal [user or login], etc.) in SQL Server, Management Studio will almost always have tooling to help. For example, right-clicking the `Tables` node under `Databases\ConferenceMessaging\` in Management Studio will give you a `New Table` menu (as well as some other specific table types that we will look at later in the book). If you want to see the script to build an existing table, right-click the table and select `Script Table As` and then `CREATE to` and choose how you want it scripted. I use these options very frequently to see the structure of a table, and I expect you will too.

WOW! eBook
www.wowebook.org

Adding Uniqueness Constraints

As I've mentioned several (or perhaps, too many) 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 about keys that are implemented with indexes:

- Adding PRIMARY KEY constraints
- Adding alternate (UNIQUE) key constraints
- Viewing uniqueness constraints
- Where other indexes fit in

Both PRIMARY KEY and UNIQUE constraints are implemented on top of unique indexes to do the enforcing of uniqueness. It's conceivable that you could use unique indexes instead of constraints, but I specifically use constraints because of the meaning that they suggest: constraints are intended to semantically represent and enforce some limitation on data, whereas indexes (which are covered in detail in Chapter 10) 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. Usually an index will be useful for performance as well, as usually when you need to enforce uniqueness, it's also the case that a user or process will be searching for a reasonably small number of values in the table.

Adding Primary Key Constraints

The first set of constraints we will add to the tables will be PRIMARY KEY constraints. The syntax of the PRIMARY KEY declaration is straightforward:

```
[CONSTRAINT constraintname] PRIMARY KEY [CLUSTERED | NONCLUSTERED]
```

As with 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<tablename>. Generally, you will want to make the primary key clustered for the table, as normally the columns of the primary key will be the most frequently used for accessing rows. This is definitely not always the case, and will usually be something discovered later during the testing phase of the project. 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.

■ **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 DFLTTopic_TopicId
                                DEFAULT(NEXT VALUE FOR  dbo.TopicIdGenerator)
                                CONSTRAINT PKTopic PRIMARY KEY,
        Name                nvarchar(30)  NOT NULL ,
        Description         varchar(60)  NOT NULL ,
```


WOW! eBook
www.wowebook.org

```
        RowCreateTime        datetime2(0)  NOT NULL ,
        RowLastUpdateTime    datetime2(0)  NOT NULL
);
```

Or, if it is 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 PKExampleKey
                PRIMARY KEY (ExampleKeyColumn1, ExampleKeyColumn2)
)
```

Another 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 (CLUSTERED is optional, but is included for emphasis):

```
ALTER TABLE Attendees.AttendeeType
     ADD CONSTRAINT PKAttendeeType PRIMARY KEY CLUSTERED (AttendeeType);

ALTER TABLE Attendees.MessagingUser
     ADD CONSTRAINT PKMessagingUser PRIMARY KEY CLUSTERED (MessagingUserId);

ALTER TABLE Attendees.UserConnection
     ADD CONSTRAINT PKUserConnection PRIMARY KEY CLUSTERED (UserConnectionId);

ALTER TABLE Messages.Message
     ADD CONSTRAINT PKMessage PRIMARY KEY CLUSTERED (MessageId);

ALTER TABLE Messages.MessageTopic
     ADD CONSTRAINT PKMessageTopic PRIMARY KEY CLUSTERED (MessageTopicId);

ALTER TABLE Messages.Topic
     ADD CONSTRAINT PKTopic PRIMARY KEY CLUSTERED (TopicId);
```

■ **Tip** Although the CONSTRAINT <constraintName> part of any constraint declaration is optional, 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 (and will be far harder to compare to multiple databases created from the same script, like for Dev, Test, and Prod). 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
```


WOW! eBook
www.wowebook.org

```
WHERE  table_schema = 'dbo'
  AND  table_name = 'TestConstraintName';
```

You see the name chosen is something hideous 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 an `UNIQUE` constraint. These are very similar to `PRIMARY KEY` constraints (other than they can have nullable columns) 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 AKMessage_TimeUserAndText UNIQUE
      (RoundedMessageTime, MessagingUserId, Text);

ALTER TABLE Messages.Topic
     ADD CONSTRAINT AKTopic_Name UNIQUE (Name);

ALTER TABLE Messages.MessageTopic
     ADD CONSTRAINT AKMessageTopic_TopicAndMessage UNIQUE
      (MessageId, TopicId, UserDefinedTopicName);

ALTER TABLE Attendees.MessagingUser
     ADD CONSTRAINT AKMessagingUser_UserHandle UNIQUE (UserHandle);

ALTER TABLE Attendees.MessagingUser
     ADD CONSTRAINT AKMessagingUser_AttendeeNumber UNIQUE
     (AttendeeNumber);

ALTER TABLE Attendees.UserConnection
     ADD CONSTRAINT AKUserConnection_Users UNIQUE
     (MessagingUserId, ConnectedToMessagingUserId);
```

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 again that the computed column you specify must be based on a deterministic expression to be used in an index. Declaring the column as `PERSISTED` is a good way to know if it is deterministic or not.

WOW! eBook
www.wowebook.org

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 are 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 that you will add to a table other than uniqueness constraints will generally 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 CONCAT(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, is_primary_key DESC, name;
```

which, for the constraints we have created so for, returns

```
object_name                 name                           primary_key  unique_constraint
--------------------------- ------------------------------ ------------ ------------------
Attendees.AttendeeType      PKAttendees_AttendeeType       1            0
Attendees.MessagingUser     PKAttendees_MessagingUser      1            0
Attendees.MessagingUser     AKAttendees_MessagingUser_A... 0            1
Attendees.MessagingUser     AKAttendees_MessagingUser_U... 0            1
Attendees.UserConnection    PKAttendees_UserConnection     1            0
Attendees.UserConnection    AKAttendees_UserConnection_... 0            1
Messages.Message            PKMessages_Message             1            0
Messages.Message            AKMessages_Message_TimeUser... 0            1
Messages.MessageTopic       PKMessages_MessageTopic        1            0
Messages.MessageTopic       AKMessages_MessageTopic_Top... 0            1
Messages.Topic              PKMessages_Topic               1            0
Messages.Topic              AKMessages_Topic_Name          0            1
```

As you start to do index tuning, one of the major tasks is to determine whether indexes are being used and to 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.

WOW! eBook
www.wowebook.org

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 a function, even a user-defined one that accesses a table. 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 DFLTMessagingUser_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 CONCAT('ALTER TABLE ',TABLE_SCHEMA,'.',TABLE_NAME,CHAR(13),CHAR(10),
               '    ADD CONSTRAINT DFLT', TABLE_NAME, '_' ,
              COLUMN_NAME, CHAR(13), CHAR(10),
       '    DEFAULT (SYSDATETIME()) FOR ', COLUMN_NAME,';')
FROM   INFORMATION_SCHEMA.COLUMNS
WHERE  COLUMN_NAME in ('RowCreateTime', 'RowLastUpdateTime')
  and  TABLE_SCHEMA in ('Messages','Attendees')
ORDER BY TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME;
```


WOW! eBook
www.wowebook.org

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)

Adding relationships is perhaps the most tricky of the constraints because both the parent and child tables need to exist to create the constraint. Hence, it is more common to add foreign keys using the `ALTER TABLE` statement, but you can also do this using the `CREATE TABLE` statement.

WOW! eBook
www.wowebook.org

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 a pretty rare 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:

```
ALTER TABLE TableName [WITH CHECK | WITH NOCHECK]
  ADD [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> ];
```

The components of this syntax are as follows:

- <referenceTable>: The parent table in the relationship.
- <referenceColumns>: 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, the child key will be set 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, 98.934% 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 common 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 it for relationships like Customer <-- Invoice. Deleting a customer who has invoices as a general rule is probably not desired. So you will want the client code to specifically delete the invoices before deleting the customer on the rare occasion that is desired to be deleted.

Note too the optional [WITH CHECK | WITH NOCHECK] specification. When you create a constraint, the WITH NOCHECK setting (the default) gives you the opportunity to create the constraint without checking existing data. Using NOCHECK and leaving the values unchecked is a generally bad thing to do because if you try to resave the exact same data that existed in a row, you could get an error. Also, if the constraint is built using WITH CHECK, the query optimizer can possibly make use of this fact when building a query plan.

Thinking back to our modeling, there were optional and required relationships such as the one in Figure 6-29.

WOW! eBook
www.wowebook.org

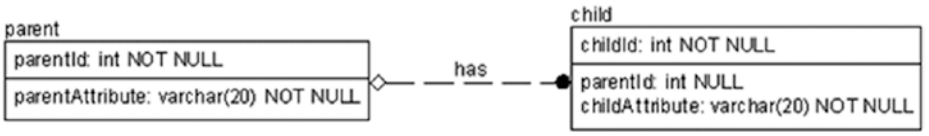


Figure 6-29. *Optional parent-to-child relationship requires NULL on the migrated key*

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.

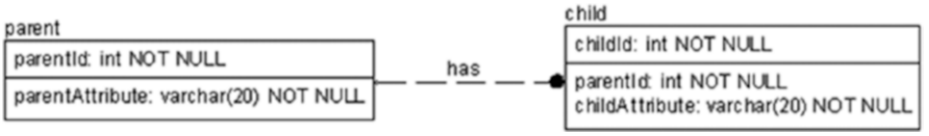


Figure 6-30. *Required parent-to-child relationship requires NOT NULL on the migrated key*

This is all you need to do, because SQL Server knows that when the referencing key allows a `NULL`, the relationship value is optional. In our model, represented in Figure 6-31, we have seven relationships modeled.

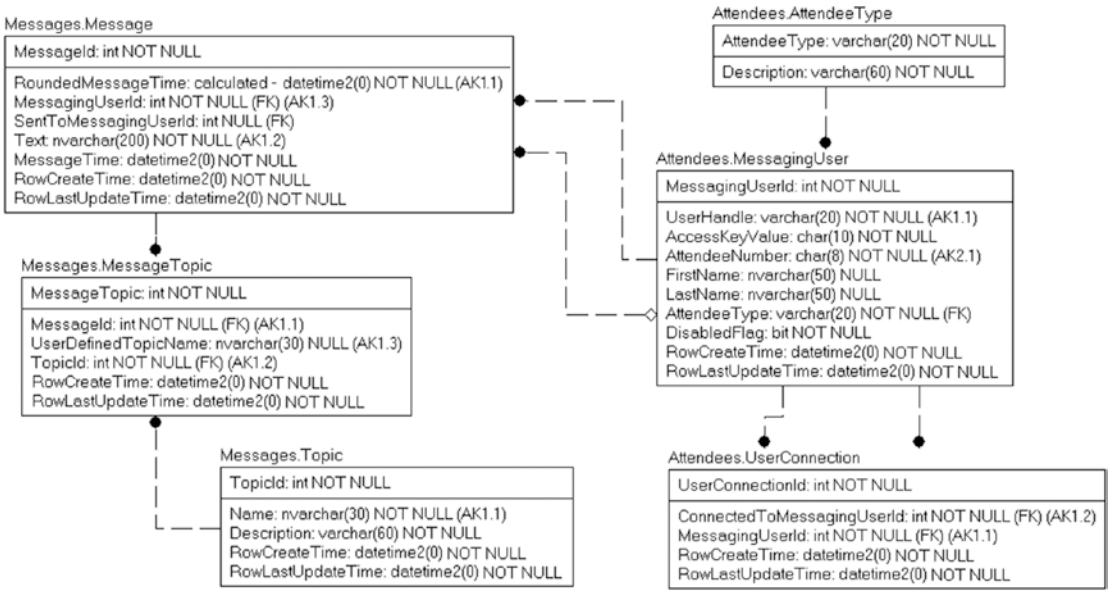


Figure 6-31. *Messaging model for reference*

WOW! eBook
www.wowebook.org

Recall from Figure 6-9 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:

```
FKMessagingUser$IsSent$Messages_Message
```

Doing this 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` and `MessagingUser`. Since it uses a natural key, it is a target for the `UPDATE CASCADE` option. However, note that if you have a lot of `MessagingUser` rows, this operation can be 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 FKMessagingUser$IsSent$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` table 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` table. Hence, you might consider implementing both of these as `DELETE CASCADE`. However, if you execute the following statements:

```
ALTER TABLE Attendees.UserConnection
        ADD CONSTRAINT
          FKMessagingUser$ConnectsToUserVia$Attendees_UserConnection
        FOREIGN KEY (MessagingUserId) REFERENCES Attendees.MessagingUser(MessagingUserId)
        ON UPDATE NO ACTION
        ON DELETE CASCADE;

ALTER TABLE Attendees.UserConnection
        ADD CONSTRAINT
          FKMessagingUser$IsConnectedToUserVia$Attendees_UserConnection
        FOREIGN KEY  (ConnectedToMessagingUserId)
                            REFERENCES Attendees.MessagingUser(MessagingUserId)
        ON UPDATE NO ACTION
        ON DELETE CASCADE;
```

```
Introducing FOREIGN KEY constraint 'FKMessagingUser$IsConnectedToUserVia$Attende
es_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.
```

WOW! eBook
www.wowebook.org

Basically, this message 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 (which I will do as an example).

I will also note that, in many ways, having these limitations on cascading operations 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.

I will change the constraints to NO ACTION and re-create (dropping the one that was created first):

```
ALTER TABLE Attendees.UserConnection
        DROP CONSTRAINT
          FKMessagingUser$ConnectsToUserVia$Attendees_UserConnection;
GO

ALTER TABLE Attendees.UserConnection
        ADD CONSTRAINT
          FKMessagingUser$ConnectsToUserVia$Attendees_UserConnection
        FOREIGN KEY (MessagingUserId) REFERENCES Attendees.MessagingUser(MessagingUserId)
        ON UPDATE NO ACTION
        ON DELETE NO ACTION;

ALTER TABLE Attendees.UserConnection
        ADD CONSTRAINT
          FKMessagingUser$IsConnectedToUserVia$Attendees_UserConnection
        FOREIGN KEY  (ConnectedToMessagingUserId)
                            REFERENCES Attendees.MessagingUser(MessagingUserId)
        ON UPDATE NO ACTION
        ON DELETE NO ACTION;
GO
```

And the following INSTEAD OF trigger will go ahead and delete the rows before the actual operation that the user tried is executed:

```
CREATE TRIGGER MessagingUser$InsteadOfDeleteTrigger
ON Attendees.MessagingUser
INSTEAD OF DELETE 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 int = (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
```


WOW! eBook
www.wowebook.org

```
   BEGIN TRY
          --[validation section]
          --[modification section]

          --implement multi-path cascade delete in trigger
          DELETE FROM Attendees.UserConnection
          WHERE  MessagingUserId IN (SELECT MessagingUserId FROM DELETED);

          DELETE FROM Attendees.UserConnection
          WHERE  ConnectedToMessagingUserId IN (SELECT MessagingUserId FROM DELETED);

          --<perform action>
          DELETE FROM Attendees.MessagingUser
          WHERE  MessagingUserId IN (SELECT MessagingUserId FROM DELETED);
   END TRY
   BEGIN CATCH
          IF @@trancount > 0
              ROLLBACK TRANSACTION;
          THROW;
     END CATCH;
END;
```

For the two relationships between `MessagingUser` and `Message`, it may seem like we want to use cascade operations, but in this case, since we implemented a disabled indicator in the `MessagingUser` table, we would probably not use cascade operations. If the `MessagingUser` had not created message rows yet, it could be deleted, otherwise it would usually be disabled (and the row would not be deleted, which would cause the previous trigger to fail and leave the connection rows alone too).

If a system administrator wants to remove the user completely, a module would likely be created to manage this operation, as it would be the exception rather than the rule. So for this example, we will implement `NO ACTION` on `DELETE`:

```
ALTER TABLE Messages.Message
        ADD CONSTRAINT FKMessagingUser$Sends$Messages_Message FOREIGN KEY
            (MessagingUserId) REFERENCES Attendees.MessagingUser(MessagingUserId)
        ON UPDATE NO ACTION
        ON DELETE NO ACTION;

ALTER TABLE Messages.Message
        ADD CONSTRAINT FKMessagingUser$IsSent$Messages FOREIGN KEY
            (SentToMessagingUserId) REFERENCES Attendees.MessagingUser(MessagingUserId)
        ON UPDATE NO ACTION
        ON DELETE NO ACTION;
```

Whether to use cascading operations should not be considered lightly. `NO ACTION` relationships prevent mistakes from causing calamity. `CASCADE` operations (including `SET NULL` and `SET DEFAULT`, which give you additional possibilities for controlling cascading operations), are powerful tools to make coding easier, but could wipe out a lot of data. For example, if you case cascade from `MessagingUser` to `Message` and `UserConnection`, then do `DELETE MessagingUser`, data can be gone in a hurry.

WOW! eBook
www.wowebook.org

The next relationship we will deal with is between Topic and MessageTopic. We don't want Topics to be deleted once set up and 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
           FKTopic$CategorizesMessagesVia$Messages_MessageTopic FOREIGN KEY
             (TopicId) REFERENCES Messages.Topic(TopicId)
        ON UPDATE NO ACTION
        ON DELETE NO ACTION;
```

The next-to-last relationship to implement is the MessageTopic to Message relationship. Just like the Topic to MessageTopic relationship, there is no need to automatically delete messages if the topic is deleted.

```
ALTER TABLE Messages.MessageTopic
        ADD CONSTRAINT FKMessage$isCategorizedVia$MessageTopic FOREIGN KEY
            (MessageId) REFERENCES Messages.Message(MessageId)
        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 (or different engine models, in that a memory-optimized table's foreign keys cannot reference an on-disk table's constraints). When this situation occurs, these relationship types need to be implemented via triggers from the on-disk table, or interpreted T-SQL stored procedures for in-memory tables.

While cross-container (as the memory-optimized and on-disk storage/transaction engines are commonly referred to) relationships may be useful at times (more on this in later chapters), it is definitely considered 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>
```


WOW! eBook
www.wowebook.org

One thing interesting about CHECK constraints is how the <BooleanExpression> is evaluated. 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.)

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. 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 NULLs if another column has a given value. 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 (NameIsNotNullFlag = 0))`.

Note that the [WITH CHECK | WITH NOCHECK] specification works just like it did for FOREIGN KEY constraints

In our model, we had two domain predicates 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 an 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 column from the Messages.MessageTopic table:

```
ALTER TABLE Messages.Topic
   ADD CONSTRAINT CHKTopic_Name_NotEmpty
       CHECK (LEN(RTRIM(Name)) > 0);
```


WOW! eBook
www.wowebook.org

```
ALTER TABLE Messages.MessageTopic
   ADD CONSTRAINT CHKMessageTopic_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

To implement this domain, things get a bit more interesting:

```
ALTER TABLE Attendees.MessagingUser
  ADD CONSTRAINT CHKMessagingUser_UserHandle_LengthAndStart
     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 have 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 CHKMessageTopic_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 predicate, 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).

WOW! eBook
www.wowebook.org

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 columns 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. Automatically generated values are not limited to implementation columns like these Row% prefixed columns, but most often, we are implementing them strictly for software's sake, hence the reason that we will implement them in such a manner that the client cannot modify the values. A big difference between implementation columns and user-facing columns is that we will code our triggers to ensure that the data in the column cannot be overridden without disabling the trigger.

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 set in the row, since the actual insert will be done in the trigger, outside of the scope of the code. @@IDENTITY will work, but it has its own issues, particularly with triggers that perform cascading operations.
- An OUTPUT clause on a modification DML statement 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 consider using a SEQUENCE-based key, or use one of the natural keys you have implemented to get the inserted value if you are inserting a single row. I only made one table use a SEQUENCE because so many tables will use an IDENTITY column for surrogate keys because they are easy to implement.

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 briefly mention about triggers is multirow operations. Well-written triggers take into consideration that any INSERT, UPDATE, or DELETE statement may affect multiple rows in one execution. 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
```


WOW! eBook
www.wowebook.org

```
   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

  END CATCH
END
```

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 GETDATE():

```
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;
```


WOW! eBook
www.wowebook.org

```
   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, particularly due to the loss of SCOPE_IDENTITY(), 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 may not be allowed is if you have a cascade operation. For example, consider our relationship fromMessgingUser to AttendeeType:

```
ALTER TABLE Attendees.MessagingUser
    ADD  CONSTRAINT FKMessagingUser$IsSent$Messages_Message
            FOREIGN KEY(AttendeeType)
            REFERENCES Attendees.AttendeeType (AttendeeType)
ON UPDATE CASCADE;
```

Since this is a 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
```


WOW! eBook
www.wowebook.org

```
          @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 triggers for all of the tables in our database. They will follow the same basic 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.

Documenting Your Database

In your modeling process, 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. A great way to share this data is using extended properties that allow you to store specific information about objects. 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

WOW! eBook
www.wowebook.org

- 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. The datatype is `sql_variant`, with a maximum length limitation of 7500.
- `@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.

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

WOW! eBook
www.wowebook.org

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',
   @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.

WOW! eBook
www.wowebook.org

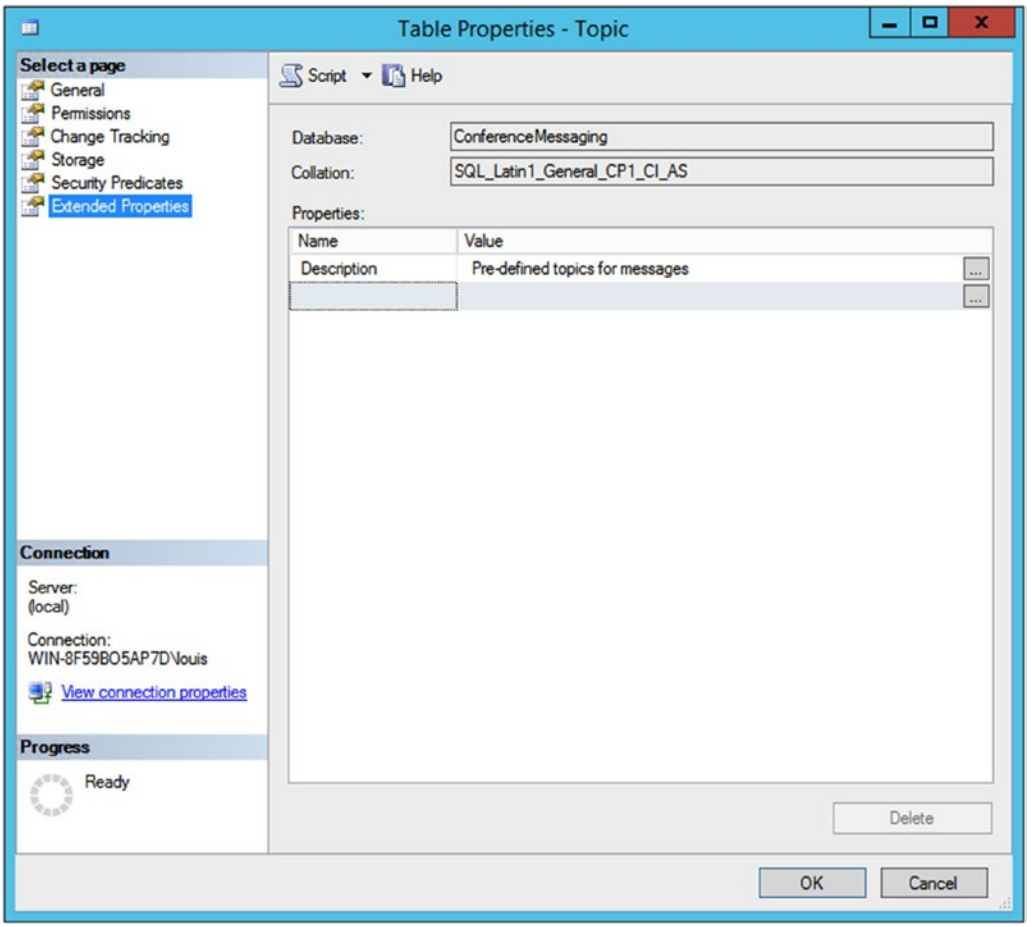


Figure 6-32. *Descriptions in Management Studio (reward for hard work done)*

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
FROM   fn_listExtendedProperty ( 'Description',
                                 'Schema','Messages',
                                 'Table','Topic',
                                 'Column',null);
```


WOW! eBook
www.wowebook.org

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 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 the 7,500-byte limitation). 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, columns, and schemas either. Constraints, databases, and many other objects in the database can have extended properties.

■ **Note** As of SQL Server 2016, extended properties are not available on memory-optimized tables.

Viewing the Basic System 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 is a plethora of `sys` schema objects. However, they can be a bit messier to use and aren't based on a standard that is generally applied to SQL Server, Oracle, etc., 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 use the `sys` schema objects (commonly referred to as the system catalog) to get metadata as well.

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 tools automatically creates a schema for every user that gets created.

```
SCHEMA_NAME     SCHEMA_OWNER
--------------- ----------------
Messages        dbo
Attendees       dbo
```


WOW! eBook
www.wowebook.org

■ **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` view has a lot of niceties that are useful, such as returning readable names for all of the layers of database structure I am interested in, 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 length
        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;
```

In the database that we have been working on throughout the chapter, the preceding 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)        (sysdatetime())
```


WOW! eBook
www.wowebook.org

```
Attendees.MessagingUser     datetime2(0)        (sysdatetime())
Attendees.UserConnection    int                 NULL
Attendees.UserConnection    int                 NULL
Attendees.UserConnection    int                 NULL
Attendees.UserConnection    datetime2(0)        (sysdatetime())
Attendees.UserConnection    datetime2(0)        (sysdatetime())
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)        (sysdatetime())
Messages.Message            datetime2(0)        (sysdatetime())
Messages.MessageTopic       int                 NULL
Messages.MessageTopic       int                 NULL
Messages.MessageTopic       nvarchar(30)        NULL
Messages.MessageTopic       int                 NULL
Messages.MessageTopic       datetime2(0)        (sysdatetime())
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)        (sysdatetime())
Messages.Topic              datetime2(0)        (sysdatetime())
```

To see the constraints we have added to these objects (other than defaults, which were included in the previous results), use this code:

```
SELECT TABLE_SCHEMA, 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 `CONSTRAINT_NAME` column truncated for some of the results to fit the data in):

```
TABLE_SCHEMA   TABLE_NAME       CONSTRAINT_NAME       CONSTRAINT_TYPE
-------------- ---------------- --------------------- ---------------
Attendees      AttendeeType     PKAttendeeType        PRIMARY KEY
Attendees      MessagingUser    PKMessagingUser       PRIMARY KEY
Attendees      MessagingUser    AKMessagingUser_Us... UNIQUE
Attendees      MessagingUser    AKMessagingUser_At... UNIQUE
Attendees      MessagingUser    CHKMessagingUser_U... CHECK
Attendees      MessagingUser    FKMessagingUser$Is... FOREIGN KEY
Attendees      UserConnection   FKMessagingUser$Co... FOREIGN KEY
Attendees      UserConnection   FKMessagingUser$Is... FOREIGN KEY
Attendees      UserConnection   AKUserConnection_U... UNIQUE
Attendees      UserConnection   PKUserConnection      PRIMARY KEY
Messages       Message          PKMessage             PRIMARY KEY
```


WOW! eBook
www.wowebook.org

```
Messages       Message          AKMessage_TimeUser... UNIQUE
Messages       MessageTopic     AKMessageTopic_Top... UNIQUE
Messages       MessageTopic     FKTopic$Categorize... FOREIGN KEY
Messages       MessageTopic     FKMessage$isCatego... FOREIGN KEY
Messages       MessageTopic     PKMessageTopic        PRIMARY KEY
Messages       MessageTopic     CHKMessageTopic_Us... CHECK
Messages       MessageTopic     CHKMessageTopic_Us... CHECK
Messages       Topic            CHKTopic_Name_NotE... CHECK
Messages       Topic            PKTopic               PRIMARY KEY
Messages       Topic            AKTopic_Name          UNIQUE
```

Doing this will minimally help you get an idea of what tables you have created 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_desc = 'OBJECT_OR_COLUMN' --DML trigger on a table or view
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
```


WOW! eBook
www.wowebook.org

This will return

```
TABLE_NAME                CONSTRAINT_NAME                          CHECK_CLAUSE
------------------------- ---------------------------------------- ----------------------
Messages.MessageTopic     CHKMessageTopic_UserDefinedTopicNa...  (len(rtrim([UserDefined...
Messages.MessageTopic     CHKMessageTopic_UserDefinedTopicNa...  ([UserDefinedTopicName]...
Attendees.MessagingUser   CHKMessagingUser_UserHandle_Length...  (len(rtrim([UserHandle]...
Messages.Topic            CHKTopic_Name_NotEmpty                  (len(rtrim([Name]))>(0))
```

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 screenshots of SSMS or Data Tools.

■ **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

Coding tables is a lot of fun, but it is by no means the end of the process. Now that you have structures created, you need to create test scripts to insert good and bad data into your tables to make sure that what you expect to occur does. 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 their 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 procedure, one constraint, etc.

In this section, I will give you a simplistic version of testing your database structures that you can easily do for free. In it, I will use a single script that will run and 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 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 inserts 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.

WOW! eBook
www.wowebook.org

■ **Note** A comprehensive discussion of testing is out of scope for this book because complete testing requires involvement of the entire team. The unit test is generally the first of several testing steps, which include 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 our database structures, the goal will be simply to prove to ourselves that the code we created does the minimum that it is programmed to do.

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 scenarios. Those we expect to work, and those we expect to fail. For the scenarios 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. The tests follow the pattern:

```
<statement to test>

if @@ROWCOUNT <> N 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 `THROW` 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',1;
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 make inter-table 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 Objects.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
```


WOW! eBook
www.wowebook.org

```
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]
           ,[FirstName],[LastName],[AttendeeType]
           ,[DisabledFlag])
VALUES ('FredF','0000000000','00000000','Fred','Flintstone','Regular',0);

IF @@ROWCOUNT <> 1 THROW 50000,'Attendees.MessagingUser Single Row  Failed',16;
```

Next, I 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',1;
END TRY
BEGIN CATCH
        IF ERROR_MESSAGE() NOT LIKE
                              '%CHKMessagingUser_UserHandle_LengthAndStart%'
                THROW 50000,'Check Messages.Topic.Name didn''t work',1;
END CATCH;
```

When you execute this batch, 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 nonalphanumeric 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',1;
END TRY
BEGIN CATCH
        IF ERROR_MESSAGE() NOT LIKE
                          '%CHKMessagingUser_UserHandle_LengthAndStart%'
                THROW 50000,'Check Messages.Topic.Name didn''t work',16;
END CATCH;
```


WOW! eBook
www.wowebook.org

■ **Tip** 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 get no output other than 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.

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.) To test this, I will enter a row into the table and then immediately enter another at exactly the same time. 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'
         ,SYSDATETIME());

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
                           --Row1
                           ((SELECT MessagingUserId FROM Attendees.MessagingUser
                             WHERE UserHandle = 'FredF')
                           ,(SELECT MessagingUserId FROM Attendees.MessagingUser
                             WHERE UserHandle = 'WilmaF')  --
                           ,'It looks like I will be late tonight'
                           ,SYSDATETIME()),
                           --Row2
                           ((SELECT MessagingUserId FROM Attendees.MessagingUser
                             WHERE UserHandle = 'FredF')
                           ,(SELECT MessagingUserId FROM Attendees.MessagingUser
                             WHERE UserHandle = 'WilmaF')  --
```


WOW! eBook
www.wowebook.org

```
                        ,'It looks like I will be late tonight'
                        ,SYSDATETIME());
        THROW 50000,'No error raised',1;
END TRY
BEGIN CATCH
        IF ERROR_MESSAGE() NOT LIKE '%AKMessage_TimeUserAndText%'
                THROW 50000,'Unique Message Error didn''t work (check times)',1;
END CATCH;
```

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 for this database. 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();

--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))));

IF NOT EXISTS (SELECT * FROM  Messages.Topic WHERE Name = 'General Topic')
    INSERT INTO Messages.Topic(Name, Description)
    VALUES('General Topic','General Topic');

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');
```


WOW! eBook
www.wowebook.org

```
  COMMIT TRANSACTION;
END TRY
BEGIN CATCH
        IF ERROR_MESSAGE() NOT LIKE
                          '%CHKMessageTopic_UserDefinedTopicName_NullUnlessUserDefined%'
                THROW 50000,'UserDefined Message Check Failed',1;
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 several hours to create this one for this simple six-table database. In reality, I usually start with a simple script that creates data only without testing the CHECK constraints, uniqueness constraints, and so on, because it is rare that I 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 with the people you work with/against. 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 by myself, except that I was testing 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.

- *Understand the relational engines and choose wisely*: With the classic On-Disk engine model and the In-Memory model coexisting in SQL Server, you have more choices than ever to produce highly concurrent database solutions. The vast majority of relational database projects do not need the In-Memory model, but for those that do, you can get tremendous performance for some or all of the tables in your database.
- *Invest in database generation tools*: Do this after you know what the tool should be doing. 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 will use singular names for tables and columns.

WOW! eBook
www.wowebook.org

- *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* `uniqueness` *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 `uniqueness` 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 messed up 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 implementation, but good testing is essential to know that your design works.

Deployment Lifecycle

As you complete the DDL and complete as much of the unit testing as you can, usually there are other people who are waiting to build UIs, reports, object models, etc. So when you arrive at that point in the dev process that your DDL is ready to go, things get real.

The next step is to provide a real database for your teammates to work with, one that you don't drop and re-create over and over. There are two steps to deploying a database script:

- *Brand new database*: This is easy. Simply use the script you used for testing. Sadly, this usually only works once per database, out of the 100s to 1000s of times you will need the following step.
- *Subsequent builds of the database*: Databases are different from most other types of software programming because tables have state information. If you change the design, you have to retrofit the data into the new objects. The method I have adopted is to generate the database that I test with, and consider it the "model" database. No code, no data (other than model/domain data), just what was in the design. From there I use a comparison tool to generate the differences between the model and the live database, verify the script, and apply.

We need to cover just a bit about the second step of the process. What is described is rather simplified from the myriad of possible methods of applying changes. I use the mentioned process of creating a model database, and then I compare it to a database that is mirrored in source control using Redgate's SQL Source Control tool and apply the changes. (Microsoft's Data Tools has similar functionality as part of the tools

WOW! eBook
www.wowebook.org

as well.) After checking in the changes, this is used with Redgate's SQL Compare tools to apply structure changes to multiple developer servers, and then to dev, test, and finally prod servers.

SQL Server Data Tools has lifecycle steps built in, and that tool works a lot more like Visual Studio programmers are used to, if that suits you. And other companies have other tools to help manage the database-changes life cycle.

The primary idea that I am advocating here in this section is that you keep a pristine copy of your database that you create from a modeling tool or script set, and then run your unit tests on it. From there, figure out a method of getting your changes to the other members of your team that works for you.

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.

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 and creating tables, adding constraints and modifying columns, and even creating triggers to manage automatically maintained columns. General guidelines were given for naming tables, columns, indexes, and 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 uniqueness constraints. 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 rest of the book, we will continually expand on the topics of this chapter, building more and more complex objects, and digging deeper into the behaviors of SQL Server that will take your designs from the simplicity of this chapter and allow you to create complex, high-performance database solutions.

WOW! eBook
www.wowebook.org

CHAPTER 7

Expanding Data Protection with Check Constraints and Triggers

Safety is something that happens between your ears, not something you hold in your hands.

—Jeff Cooper, US Marine, Creator of the modern technique of firing a handgun

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. I am for this also, as long as it can be completely trustworthy. When the table itself says no bad data, you can be sure that it contains no bad data (as much as you have designed what "bad data" means). And if the user interface has to duplicate the rules that have been specified in the requirements, that makes perfect sense. The data layer's error messaging is atrocious, even using a few techniques to map error messages to descriptions. We don't live in a mainframe, batch-processing world, but there are many things that the data layer can handle perfectly that external code cannot (certainly without holding locks on all of the data in the database, that is).

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 should be, the data duplications start to occur. Ultimately, user perception is governed by the reliability and integrity of the data that 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.

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. We started in the previous chapter with foreign keys, unique constraints, and a few check constraints. In this chapter, I will take it farther to show some deeper examples of what can be done when the need arises.

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_7

WOW! eBook
www.wowebook.org

One argument that arises regarding the concept of putting code in multiple locations (e.g., checking for a positive value both in the data entry blank and with a check constraint) is that it's both

- Bad for performance
- More work

As C.S. Lewis had one of his evil characters in *The Screwtape Letters* 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 from one perspective, but these two arguments miss the point. The real problem we must solve is that data can come from multiple locations:

- Users using custom, very well-built 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, even the DBA fixing data who is very careful.

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 the single location helps prevent programmers from forgetting to enforce a rule in one situation, even if they remember *everywhere* else. If you receive an error (like you might if you try to store something illogical like 'A' + 20 and expect to get back an integer anyhow), you know that the UI needs to cover a scenario that it didn't.

What's more, 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 microseconds 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 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 present two 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.

In SQL Server 2016, the Native compilation model that works with in-memory objects supports check constraints and triggers. Triggers can also be written using CLR-based objects. In this chapter I will focus only on on-disk model objects. In Chapter 13, when we discuss creating code to access the tables we have created, I will cover some of the differences in code for interpreted and native modules, and the difference is great and figures into what can be coded in triggers and `CHECK` constraints. Native compilation will improve version to version (and should be expected to make continual improvements in the Azure DB product), but as of 2016, it is still quite limited. In the downloadable Appendix B, I will include a section on writing in-memory triggers.

WOW! eBook
www.wowebook.org

Check Constraints

CHECK constraints are part of a class of the declarative data protection options. 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 DEFAULT constraints) 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 named constraints you add on, they are generally considered constraints.
- 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 covered 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 DML (and DDL) 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.

We'll be looking at building CHECK constraints using two methods:

- Simple expressions
- Complex expressions using user-defined functions

WOW! eBook
www.wowebook.org

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.

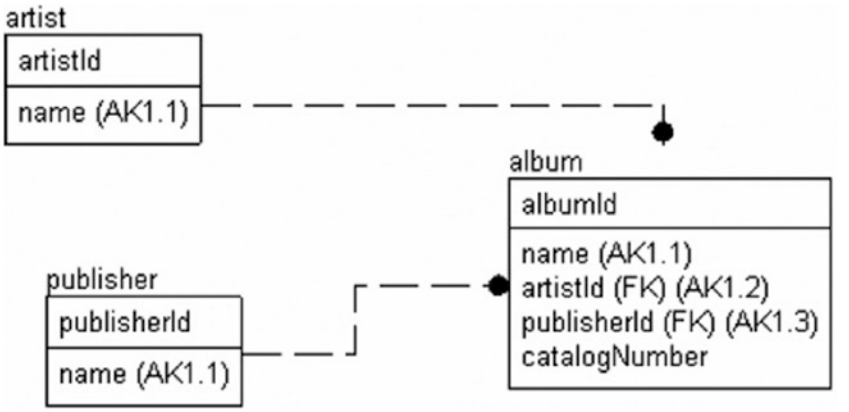


Figure 7-1. *The example schema*

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 PKArtist PRIMARY KEY CLUSTERED (ArtistId),
   CONSTRAINT PKArtist_Name UNIQUE NONCLUSTERED (Name)
);
CREATE TABLE Music.Publisher
(
        PublisherId             int CONSTRAINT PKPublisher PRIMARY KEY,
        Name                    varchar(20),
        CatalogNumberMask       varchar(100)
        CONSTRAINT DFLTPublisher_CatalogNumberMask DEFAULT ('%'),
        CONSTRAINT AKPublisher_Name UNIQUE NONCLUSTERED (Name),
);

CREATE TABLE Music.Album
(
       AlbumId int NOT NULL,
       Name varchar(60) NOT NULL,
       ArtistId int NOT NULL,
```


WOW! eBook
www.wowebook.org

```
       CatalogNumber varchar(20) NOT NULL,
       PublisherId int NOT NULL,
       CONSTRAINT PKAlbum PRIMARY KEY CLUSTERED(AlbumId),
       CONSTRAINT AKAlbum_Name UNIQUE NONCLUSTERED (Name),
       CONSTRAINT FKArtist$records$Music_Album
            FOREIGN KEY (ArtistId) REFERENCES Music.Artist(ArtistId),
       CONSTRAINT FKPublisher$Published$Music_Album
            FOREIGN KEY (PublisherId) REFERENCES Music.Publisher(PublisherId)
);
```

Then seed the data with the following:

```
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 complete 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 is 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 rule as follows (note, all examples assume a case-insensitive collation, which is almost certainly the normal):

```
ALTER TABLE Music.Artist WITH CHECK
   ADD CONSTRAINT CHKArtist$Name$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');
```


WOW! eBook
www.wowebook.org

This returns the following result:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint "CHKArtist$Name$NoPetShopNames".
The conflict occurred in database "Chapter7", table "Music.Artist", column 'Name'.
```

This keeps my music collection database safe from at least one '80s band.

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 CHKArtist$Name$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 "CHKArtist$Name$noMadonnaNa
mes". The conflict occurred in database "Chapter7", table "Music.Artist", column 'Name'.
```

Ideally, you will then change the contents of the table such that it will meet the requirements of the constraint. 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 CHKArtist$Name$noMadonnaNames
           CHECK (Name NOT LIKE '%Madonna%');
```

The statement is executed to add the CHECK constraint to the table definition, and using NOCHECK means that the invalid value does not affect the creation of the constraint. This is OK in some cases but can be very confusing because any time 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;
```


WOW! eBook
www.wowebook.org

This produces the following error message:

```
Msg 547, Level 16, State 0, Line 1
The UPDATE statement conflicted with the CHECK constraint "CHKArtist$Name$noMadonnaNames".
The conflict occurred in database "Chapter7", table "Music.Artist", column 'Name'.
```

"What?" most users will exclaim (well, unless they are the support person at 3 AM wondering what is going on, in which case they will upgrade it to "WHAT?!?!"). "If the value was in the table, shouldn't it already be good?" The user is correct. A strategy to deal with changes in format, or allowing older data to meet one criteria while new data fits a different one, can be to include a time range for the values. `CHECK Name NOT LIKE '%Madonna%' OR RowCreateDate < '20141131'` could be a reasonable compromise—as long as the users understand what is going on with their queries, naturally.

Using `NOCHECK` and leaving the values unchecked is almost worse than leaving the constraint off in many ways.

■ **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% 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 = 'CHKArtist$Name$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.

WOW! eBook
www.wowebook.org

■ **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 reasonable, 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 re-enable the constraint is too easy. For our constraint, we can try to check the values:

```
ALTER TABLE Music.Artist WITH CHECK CHECK CONSTRAINT CHKArtist$Name$noMadonnaNames;
```

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 "CHKArtist$Name$noMadonnaNa
mes". 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 executed 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 CHKArtist$Name$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 = 'CHKArtist$Name$noMadonnaNames';
```

This will return

```
definition                    is_not_trusted is_disabled
----------------------------- -------------- -----------
(NOT [Name] like '%Madonna%')  1              1
```


WOW! eBook
www.wowebook.org

Then, rerun the statement to enable the statement before we continue:

```
ALTER TABLE Music.Artist WITH CHECK CHECK CONSTRAINT CHKArtist$Name$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, such as `CHECK(LEN(ColumnName) > 0)`. This constraint is on 90% of the `character` columns in databases I design, to avoid the space character entry that drives you crazy when you don't expect it.
- *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))`.
  - The 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 needing to be a nonnegative integer: `CHECK(MilesDriven >= 0)`. This constraint is commonly needed, because there are often columns where negative values don't make sense (hours worked, miles driven, and so on), but the intrinsic type will allow it.
  - Royalty rate for an author that's less than or equal to 30%. If this rate ever could be greater, it isn't a `CHECK` constraint. So if 15% is the typical rate, the UI might warn that it isn't normal, but if 30% 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

WOW! eBook
www.wowebook.org

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.

To avoid letting a user get away with a blank column value, you can add the following constraint to prevent this from ever happening again (after deleting the two blank rows). For example, in the Album table, the Name column doesn't allow NULLs. 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.

The constraint simply 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 CHKAlbum$Name$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 "CHKAlbum$Name$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 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, no matter what the source of data.

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 it 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.

WOW! eBook
www.wowebook.org

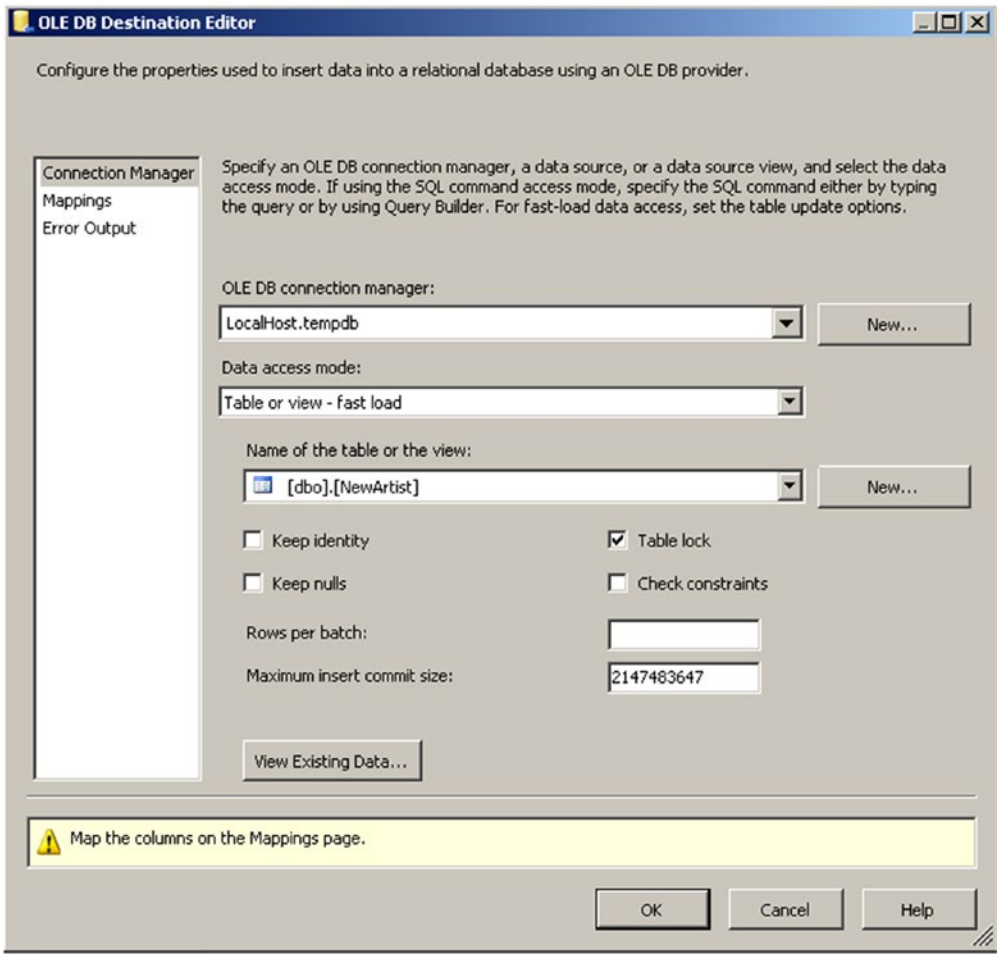


Figure 7-2. *Example SSIS OLEDB output with check constraints deselected*

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 your constraints, but in SQL Server, subqueries are not allowed. However, 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.

WOW! eBook
www.wowebook.org

However, a CHECK constraint need not be so simple. A user-defined function (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 function 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 to take what you are doing and find the best way (which you must respect) of executing the code.

Hence, it's best to try to express your Boolean expression without a UDF unless it's entirely necessary to access additional tables or to 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 --now based on the Artist ID
)

RETURNS bit
AS
BEGIN
   DECLARE @LogicalValue bit, @CatalogNumberMask varchar(100);
```


WOW! eBook
www.wowebook.org

```
   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
         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
       CHKAlbum$CatalogNumber$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
       JOIN Music.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');
```


WOW! eBook
www.wowebook.org

This causes the following error, because the catalog number of '1' doesn't match the mask set up for PublisherId number 2:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint "CHKAlbum$CatalogNumber$CatalogN
umberValidate". The conflict occurred in database "Chapter7", table "Music.Album".
```

Now, 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 match 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 might consider adding a trigger to the Publisher 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.

WOW! eBook
www.wowebook.org

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 than they will generate service desk calls every time a typical user sees 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. 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 "CHKAlbum$CatalogNumber$CatalogN
umberValidate". 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:

- `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 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).
- `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 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 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 up this value 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 ErrorHandling; --used to hold objects for error management purposes
GO
CREATE TABLE ErrorHandling.ErrorMap
(
    ConstraintName sysname NOT NULL CONSTRAINT PKErrorMap PRIMARY KEY,
    Message        varchar(2000) NOT NULL
);
```


WOW! eBook
www.wowebook.org

```
GO
INSERT ErrorHandling.ErrorMap(constraintName, message)
VALUES ('CHKAlbum$CatalogNumber$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 ErrorHandling.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   ErrorHandling.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
```


WOW! eBook
www.wowebook.org

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 ErrorHandling.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 "CHKAlbum$CatalogNumber$CatalogN
umberValidate". 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

A trigger is a type of coded module, similar to a stored procedure, that is attached to a table or view and is executed automatically when an `INSERT`, `UPDATE`, or `DELETE` statement is executed on that object. While triggers 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 any operation to another table. You can use triggers 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 Chapter 6 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 operations in a typical OLTP 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. In this chapter, I will demonstrate a few uses of triggers that can't be done automatically in SQL code any other way:

- 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

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

WOW! eBook
www.wowebook.org

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 application code is being able to access any data in the database to do the verification without sending it to a client. In Appendix B, I will discuss the mechanics of writing triggers and their various 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 chapter. 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).

Natively compiled triggers are available in SQL Server 2016, but are limited to `AFTER` triggers, as well as being subject to the current limitations on natively compiled modules.

AFTER Triggers

`AFTER` triggers fire after the DML statement has completed. Though triggers may not seem very useful, back in SQL Server 6.0 and earlier, there were no `CHECK` constraints, and even `FOREIGN KEYS` were 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 types of triggers:

- Range checks on multiple rows
- Maintaining summary values
- Cascading inserts
- Child-to-parent cascades
- 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 triggers, although not inherently terrible for performance, should be used no more than necessary.

WOW! eBook
www.wowebook.org

■ **Note** For one additional example, check the section on uniqueness 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 can't use them to validate conditions based on aggregate conditions (like `SUM(column) > 10`) because the CHECK constraint can only access data in the current row (and using a UDF, you can't see the new data either.)

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 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 objects:

```
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) NOT NULL
                  CONSTRAINT PKAccount PRIMARY KEY
        --would have other columns
);

CREATE TABLE Accounting.AccountActivity
(
        AccountNumber                char(10) NOT NULL
            CONSTRAINT FKAccount$has$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 PKAccountActivity
                      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$insertTrigger
ON Accounting.AccountActivity
AFTER INSERT AS
BEGIN
```


WOW! eBook
www.wowebook.org

```
   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 = CONCAT('Account: ', AccountNumber,
                  ' TransactionNumber:',TransactionNumber, ' for amount: ',
                  TransactionAmount,
                  ' 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;
```


WOW! eBook
www.wowebook.org

The key to 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. The base of the query is a query one might run to check the status of the table without a trigger. Is there an account where the sum of their transactions is less than 0?

```
SELECT AccountNumber
FROM Accounting.AccountActivity AS AccountActivity
GROUP BY AccountNumber
HAVING SUM(TransactionAmount) < 0;
```

Then we include a correlation to the rows that have been created since this is an insert trigger:

```
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 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. Now, as you can see up in the trigger, this is placed into the trigger using an IF EXISTS statement, which is then followed by the error handing stuff:

```
   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             --error handling stuff
```

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','A0000000000000000001','20050712',100),
       ('1111111111','A0000000000000000002','20050713',100);
```


WOW! eBook
www.wowebook.org

Now, let's see what happens when we violate this rule:

```
INSERT  INTO Accounting.AccountActivity(AccountNumber, TransactionNumber,
                                        Date, TransactionAmount)
VALUES ('1111111111','A0000000000000000003','20050713',-300);
```

Here's the result:

```
Msg 50000, Level 16, State 16, Procedure AccountActivity$insertTrigger, Line 40
Account: 1111111111 TransactionNumber:A0000000000000000003 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)
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 40
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 with a bit of work if that was an issue. 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 its worth. Processes would likely be established on how to deal with certain errors being returned.

■ **Tip** Error handling will be covered in more detail in the "Dealing with Triggers and Constraint Errors" section later in this chapter. I have used a very simple error model for these triggers that simply rethrows the error that occurs in the trigger after rolling back the transaction.

WOW! eBook
www.wowebook.org

If this is a real accounting-oriented table, then an INSERT trigger may be enough, as a true accounting system handles modifications with offsetting entries (delete a $100 charge with a -$100 charge.) However, if you did need to process deletes, you would change the base query to

```
SELECT AccountNumber
FROM Accounting.AccountActivity AS AccountActivity
WHERE EXISTS (SELECT *
              FROM   deleted
              WHERE  deleted.AccountNumber = AccountActivity.AccountNumber)
GROUP BY AccountNumber
HAVING SUM(TransactionAmount) < 0;
```

For the DELETE trigger, and for the UPDATE trigger, we need to check both tables (in case someone updates the AccountNumber, if that is allowed):

```
SELECT AccountNumber
FROM Accounting.AccountActivity AS AccountActivity
WHERE EXISTS (SELECT *
              FROM   (SELECT AccountNumber
                      FROM   deleted
                      UNION ALL
                      SELECT AccountNumber
                      FROM   inserted) AS CheckThese
              WHERE  CheckThese.AccountNumber = AccountActivity.AccountNumber)
GROUP BY AccountNumber
HAVING SUM(TransactionAmount) < 0;
```

Changeable keys such as an account number that can be modified on a transaction trips up a lot of people when implementing data checking that spans multiple rows.

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$insertTrigger';
```

Maintaining Summary Values

While in the first five chapters of the book I preached strenuously against maintaining summary values, there are cases where some form of active summarization may be necessary. For example:

- There is no other reasonable method available to optimize a process.
- The amount of reads of the summary values is far greater than the activity on the lower values and the number of times the data is modified and therefore changed.

WOW! eBook
www.wowebook.org

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 DFLTAccount_BalanceAmount DEFAULT (0.00);
```

Then, I will update the Balance column to have the current value of the data in the AccountActivity rows. First, run 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  UpdateCTE 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 = UpdateCTE.NewBalance
FROM   Accounting.Account
         JOIN UpdateCTE
               ON Account.AccountNumber = UpdateCTE.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 caused the trigger to fire as we did in the previous section using an EXISTS filter to let you not have to worry about whether one new row was created for the account or 100.

```
ALTER TRIGGER Accounting.AccountActivity$insertTrigger
ON Accounting.AccountActivity
AFTER INSERT AS
BEGIN
   SET NOCOUNT ON;
   SET ROWCOUNT 0; --in case the client has modified the rowcount
```


WOW! eBook
www.wowebook.org

```
--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]
--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)
```


WOW! eBook
www.wowebook.org

```
        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;
```

This 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);
```


WOW! eBook
www.wowebook.org

Again, the query on the AccountActivity and Account tables should show the same balances:

```
AccountNumber BalanceAmount    SummedBalance
------------- ---------------- -------------------
1111111111    400.00           400.00
2222222222    200.00           200.00
```

If you wanted an UPDATE or DELETE trigger, you would simply apply the same logic in the EXISTS expression as in the previous section.

I can't stress enough that this type of summary data strategy should be the rare exception, not the rule. But when you have to implement summary data, using a trigger to make sure it happens is a great strategy. One of the more frustrating problems that one has to deal with when using data is summary data that doesn't match the data that is the supposed source, because it takes time away from making progress with creating new software.

■ **Caution** When introducing triggers, make sure you test single-row and multirow operations, all variations of INSERTs, UPDATEs, and DELETEs, and what happens if you modify the primary key, or key you are using to group data on. In Appendix B, I will also cover some settings that determine the behavior if you have multiple triggers on the same operation that must also be tested.

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.

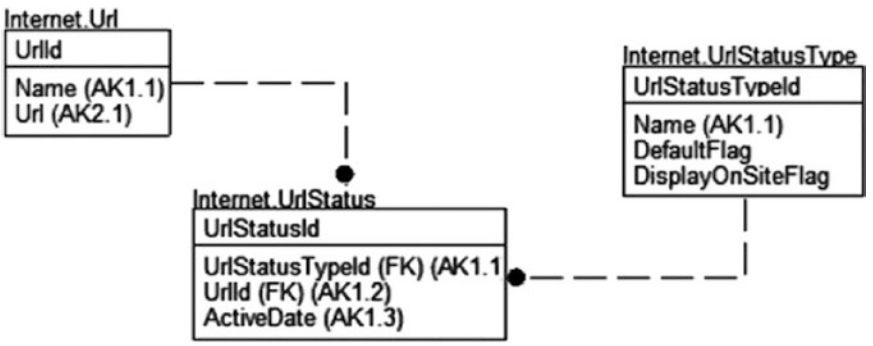


Figure 7-3. *Storing URLs for a website-linking system*

WOW! eBook
www.wowebook.org

```
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 AKUrl_Name UNIQUE,
    Url   varchar(200) NOT NULL CONSTRAINT AKUrl_Url UNIQUE
);

--Not a user manageable table, so not using identity key (as discussed in
--Chapter 6 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 PKUrlStatusType PRIMARY KEY,
        Name varchar(20) NOT NULL
                        CONSTRAINT AKUrlStatusType UNIQUE,
        DefaultFlag bit NOT NULL,
        DisplayOnSiteFlag bit NOT NULL
);

CREATE TABLE Internet.UrlStatus
(
        UrlStatusId int NOT NULL IDENTITY(1,1)
                        CONSTRAINT PKUrlStatus PRIMARY KEY,
        UrlStatusTypeId int NOT NULL
                        CONSTRAINT
               FKUrlStatusType$defines_status_type_of$Internet_UrlStatus
                        REFERENCES Internet.UrlStatusType(UrlStatusTypeId),
        UrlId int NOT NULL
          CONSTRAINT FKUrl$has_status_history_in$Internet_UrlStatus
                        REFERENCES Internet.Url(UrlId),
        ActiveTime        datetime2(3),
        CONSTRAINT AKUrlStatus_statusUrlDate
                        UNIQUE (UrlStatusTypeId, UrlId, ActiveTime)
);

--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`:

WOW! eBook
www.wowebook.org

```
CREATE TRIGGER Internet.Url$insertTrigger
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 to apply this one row to
                                          --rows in inserted
   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;
```

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 ('Author''s Website',
        'http://drsql.org');

SELECT Url.Url,Url.Name,UrlStatusType.Name as Status, UrlStatus.ActiveTime
FROM   Internet.Url
          JOIN Internet.UrlStatus
```


WOW! eBook
www.wowebook.org

```
        ON Url.UrlId = UrlStatus.UrlId
    JOIN Internet.UrlStatusType
        ON UrlStatusType.UrlStatusTypeId = UrlStatus.UrlStatusTypeId;
```

This returns the following results:

```
Url                Name                Status              ActiveTime
------------------ ------------------- ------------------- --------------------------
http://drsql.org   Author's Website    Unverified          2016-04-11 20:43:52.954
```

■ **Tip** It's easier if users can't modify the data in 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. One of the later examples in this chapter will be to force no action to be performed for a DML operation.

Cascading from Child to Parent

All the cascade operations on updates 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 that is only desired when one or more child rows exist. 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 console 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, I have a table for the `GamePlatform` (the system) and another for the actual `Game` itself. This is a many-to-many relationship, so I 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, I want to delete the game from the database. The tables are shown in Figure 7-4.

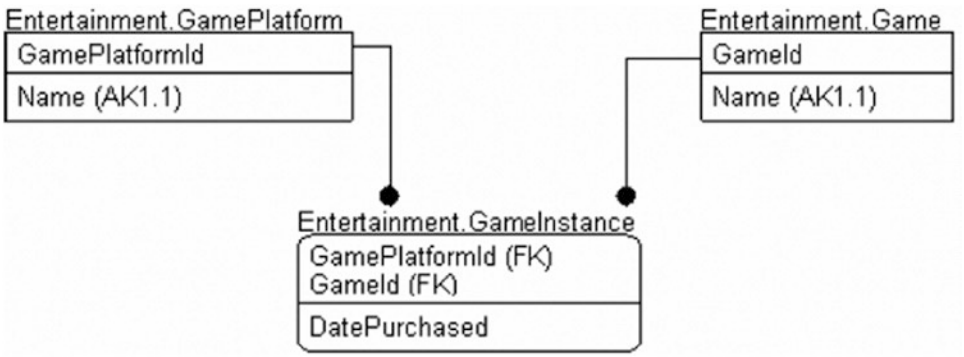


Figure 7-4. *The game tables*

WOW! eBook
www.wowebook.org

```
--start a schema for entertainment-related tables
CREATE SCHEMA Entertainment;
GO
CREATE TABLE Entertainment.GamePlatform
(
    GamePlatformId int NOT NULL CONSTRAINT PKGamePlatform PRIMARY KEY,
    Name  varchar(50) NOT NULL CONSTRAINT AKGamePlatform_Name UNIQUE
);
CREATE TABLE Entertainment.Game
(
    GameId  int NOT NULL CONSTRAINT PKGame PRIMARY KEY,
    Name    varchar(50) NOT NULL CONSTRAINT AKGame_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 PKGameInstance PRIMARY KEY (GamePlatformId, GameId),
    CONSTRAINT FKGame$is_owned_on_platform_by$EntertainmentGameInstance
                FOREIGN KEY (GameId)
                        REFERENCES Entertainment.Game(GameId) ON DELETE CASCADE,
      CONSTRAINT FKGamePlatform$is_linked_to$EntertainmentGameInstance
                FOREIGN KEY (GamePlatformId)
                        REFERENCES Entertainment.GamePlatform(GamePlatformId)
                ON DELETE CASCADE
);
```

Then, I insert a sampling of data:

```
INSERT  Entertainment.Game (GameId, Name)
VALUES (1,'Disney Infinity'),
       (2,'Super Mario Bros');

INSERT  Entertainment.GamePlatform(GamePlatformId, Name)
VALUES (1,'Nintendo WiiU'),   --Yes, as a matter of fact I am still a
       (2,'Nintendo 3DS');     --Nintendo Fanboy, why do you ask?

INSERT  Entertainment.GameInstance(GamePlatformId, GameId, PurchaseDate)
VALUES (1,1,'20140804'),
       (1,2,'20140810'),
       (2,2,'20150604');

--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
```


WOW! eBook
www.wowebook.org

```
            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 WiiU and only a single one for Nintendo 3DS:

```
Platform             Game                 PurchaseDate
-------------------- -------------------- ------------
Nintendo WiiU        Disney Infinity      2014-08-04
Nintendo WiiU        Super Mario Bros     2014-08-10
Nintendo 3DS         Super Mario Bros     2015-06-04
```

So, I create a trigger on the table to do the "reverse" cascade operation:

```
CREATE TRIGGER Entertainment.GameInstance$deleteTrigger
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);

   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 left
                          WHERE  NOT EXISTS (SELECT  *
                                             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;
```


WOW! eBook
www.wowebook.org

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.GameInstance
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 I have only a single row in the Game table:

```
platform            Game                PurchaseDate
------------------- ------------------- ------------
Nintendo 3DS        Super Mario Bros    2015-06-04
Nintendo WiiU       NULL                NULL
```

This shows us that the Game row was deleted when all instances were removed, but the platform remains. (The technique of using FULL OUTER JOINs like this will help you to be able to see permutations of matched rows. Reinsert the Game row for 'Disney Infinity' and you will see another row show up that has NULL for Platform and PurchaseDate.)

Relationships That Span Databases

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. It is more of an academic exercise at this point, as for most readers this will not occur. However, it is a good exercise, forcing you to think about all of the different changes that can go into using triggers to cover circumstances from multiple tables.

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.
- 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.

WOW! eBook
www.wowebook.org

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 if you find the need for editable keys that cascade where a `FOREIGN KEY` constraint is not allowed (which should be pretty uncommon).

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;
```

Parent Delete

Like the update, when a parent table row is deleted, we can either

- Cascade the delete to child rows
- Prevent deleting parent rows if child rows exist

WOW! eBook
www.wowebook.org

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(*)
      FROM    inserted
                 JOIN <parent> as Parent
                        ON  inserted.<parent_keys> = Parent.<parent_keys>;
```


WOW! eBook
www.wowebook.org

```
      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 column name> 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 insert trigger, just fill in the blanks. The update trigger will be identical in the code as well and could be combined if only one trigger will ever be needed. This code will not execute as is; this is just for illustration purposes.)

```
CREATE TRIGGER Contacts.Journal$insertTrigger
ON Contacts.Journal
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]
      --@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
          WHERE   inserted.CallId IS NULL;
```


WOW! eBook
www.wowebook.org

```
        --does not include null values
        SELECT  @validcount = count(*)
        FROM    inserted
                   JOIN PhoneData.Logs.Call AS Parent
                          on  inserted.CallId = Parent.CallId;

        IF @validcount + @nullcount <> @rowsAffected
          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, @Msg, 1;

           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 what would otherwise be 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 external 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 even more strongly suggest against for almost all uses of 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, much like we did in Chapter 6

WOW! eBook
www.wowebook.org

with the rowLastModifiedTime and rowCreatedTime columns. If you record last update times through client calls, it can be problematic if one of the client's clocks is a minute, a day, or even a year off. (You see this all the time in applications. My favorite example is a 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.) You can extend the paradigm of setting a value to formatting any data, such as if you wanted to make all data that was stored lowercase.

I'll demonstrate two ways you can use INSTEAD OF triggers:

- Redirecting invalid data to an exception table
- Forcing no action to be performed on a table, even by someone who technically has proper rights

It's generally a best practice not to use INSTEAD OF triggers to do error raising validations such as we did with AFTER triggers. Typically, an INSTEAD OF trigger is employed to make things happen in the background in a silent manner.

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 forms 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.
- *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,
```


WOW! eBook
www.wowebook.org

```
    ReadingTime   datetime2(3) NOT NULL
          CONSTRAINT AKWeatherReading_Date UNIQUE,
    Temperature     float NOT NULL
          CONSTRAINT CHKWeatherReading_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 ('20160101 0:00',82.00), ('20160101 0:01',89.22),
       ('20160101 0:02',600.32),('20160101 0:03',88.22),
       ('20160101 0:04',99.01);
```

As we know with CHECK constraints, this isn't going to fly:

```
Msg 547, Level 16, State 0, Line 741
The INSERT statement conflicted with the CHECK constraint "CHKWeatherReading_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 PKWeatherReading_exception PRIMARY KEY
    ReadingTime       datetime2(3) NOT NULL,
    Temperature       float NULL
);
```

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
```


WOW! eBook
www.wowebook.org

```
          @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 150);

           --GOOD data
          INSERT Measurements.WeatherReading (ReadingTime, Temperature)
          SELECT ReadingTime, Temperature
          FROM   inserted
          WHERE  (Temperature BETWEEN -80 and 150);
   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 ('20160101 0:00',82.00), ('20160101 0:01',89.22),
       ('20160101 0:02',600.32),('20160101 0:03',88.22),
       ('20160101 0:04',99.01);

SELECT *
FROM Measurements.WeatherReading;
```

The good data is in the following output:

```
WeatherReadingId ReadingTime              Temperature
---------------- ------------------------ ----------------------
4                2016-01-01 00:00:00.000  82
5                2016-01-01 00:01:00.000  89.22
6                2016-01-01 00:03:00.000  88.22
7                2016-01-01 00:04:00.000  99.01
```


WOW! eBook
www.wowebook.org

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.

One note about INSTEAD OF triggers. If you are doing a singleton insert, you may be inclined to use SCOPE_IDENTITY() to fetch the row that is inserted. If you have an INSTEAD OF trigger on the table, this will not give you the desired value:

```
INSERT  INTO Measurements.WeatherReading (ReadingTime, Temperature)
VALUES ('20160101 0:05',93.22);

SELECT SCOPE_IDENTITY();
```

This returns NULL. If you need the identity value, you can use the alternate key for the row (in this case the ReadingTime), or use a SEQUENCE generator as we did in Chapter 6 for one of the tables which provides control over the value that is used.

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)
);
INSERT  INTO System.Version (DatabaseVersion)
VALUES ('1.0.12');
```


WOW! eBook
www.wowebook.org

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
```


WOW! eBook
www.wowebook.org

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';
```

Check the data again,

```
SELECT *
FROM   System.Version;
```

and 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 Trigger and Constraint 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. 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, and will depend on what you want to occur.

Take a TRY-CATCH block, such as this one:

```
BEGIN TRY
        <DML STATEMENT>
END TRY
BEGIN CATCH
        <handle it>
END CATCH;
```


WOW! eBook
www.wowebook.org

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 the rare use of CLR triggers, you're in charge of whether the connection ends. When a CHECK constraint causes the error or executes a simple THROW or RAISERROR, you'll be in a transaction. Generically, here's the CATCH block that I use (as I have used in the triggers written so far in this chapter):

```
    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. It is simpler to do as a rule, and is 99.997% of the time what will be desired. In order to show the different scenarios that can occur, I will build the following abstract tables for demonstrating trigger and constraint error handling:

```
CREATE SCHEMA alt;
GO
CREATE TABLE alt.errorHandlingTest
(
    errorHandlingTestId   int CONSTRAINT PKerrorHandlingTest PRIMARY KEY,
    CONSTRAINT CHKerrorHandlingTest_errorHandlingTestId_greaterThanZero
           CHECK (errorHandlingTestId > 0)
);
```

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$insertTrigger
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;
```

The first thing to understand is that when a normal constraint causes the DML operation to fail, the batch will continue to operate:

```
--NO Transaction, Constraint Error
INSERT alt.errorHandlingTest
VALUES (-1);
SELECT 'continues';
```


WOW! eBook
www.wowebook.org

You will see that the error is raised, and then the SELECT statement is executed:

```
Msg 547, Level 16, State 0, Line 913
The INSERT statement conflicted with the CHECK constraint "CHKerrorHandlingTest_
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
```

This fairly elegant stoppage occurs using THROW because THROW stops the batch. However, using RAISERROR the batch will still stop, but it will give you a message about stopping the trigger. As I will show, if you just THROW an error in the trigger, it will still end up ending as well, in a far less elegant manner.

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;
```


WOW! eBook
www.wowebook.org

This returns the following:

```
XACT_STATE        ErrorNumber ErrorMessage
----------------- ----------- ------------------------------------------------------------
Committable       547         The INSERT statement conflicted with the CHECK constraint...
```

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 (and not entirely obvious to the programmer). 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 variable 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$insertTrigger
ON alt.errorHandlingTest
AFTER INSERT
AS
```


WOW! eBook
www.wowebook.org

```
    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 referred to as a *doomed* transaction. An uncommittable transaction is still in force but can never be committed and must eventually 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.

WOW! eBook
www.wowebook.org

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 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 = CONCAT( COALESCE(@errorMessage,''), ' ( System Error: ',
                                ERROR_NUMBER(),':',ERROR_MESSAGE(),
                                ' : Line Number:',ERROR_LINE());
        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. In Appendix B, I outline some additional methods you might take to log errors, using a stored procedure I call `ErrorHandling.ErrorLog$insert`, depending on whether the error is something that you expected to occur on occasion or is something (as I said about triggers) that really shouldn't happen. If I was implementing the code in such a way that I expected errors to occur, I might also include a call to something like the `ErrorHandling.ErrorMap$MapError` procedure that was discussed earlier to beautify the error message value for the system error.

Error handling is definitely a place where SQL Server's T-SQL language lacks in comparison to almost any other language, but took leaps of improvement in 2005 with `TRY…CATCH`, with a few improvements continued in 2012 with the ability to rethrow an error using `THROW`, which we have used in the standard trigger template. Of course, the most important part of writing error-handing code is the testing you do to make sure that it works!

WOW! eBook
www.wowebook.org

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 almost any datatype. It sounds silly perhaps, but it is possible. But you know better after reading Chapter 6 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, possibly 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 table's 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.

Or course, not all data protection can be done at the object level, and some will need to be managed using client code or even asynchronous processes that execute long after the data has been entered. 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. It also is far better in terms of dealing with concurrent users making frequent changes to data.

Summary

Now, you've finished the task of developing the data storage for your databases. If you've planned out your data storage and data protection layer, 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 "IdiotWhoInsultedMe"—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.

WOW! eBook
www.wowebook.org

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 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.

Use `CHECK` constraints to protect data in a single row. You can access any column in the table, but only the data in that one row. You can access data in other tables using user-defined functions, if you so desire, but note that just like building your own foreign key checking, you need to consider what happens in the table you are checking, and the table you are referencing. Use triggers to do most complex checking, especially when you need to reference rows in the same table, like checking balances. Triggers can also introduce side effects, like maintaining denormalized data if it is needed, or even calling other procedures like `sp_db_sendmail`. The most important part of writing a trigger is understanding that it executes once per DML execution, so once no matter if 1 row was affected, or 1 billion rows. Plan ahead.

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.

WOW! eBook
www.wowebook.org

CHAPTER 8

Patterns and Anti-Patterns

In short, no pattern is an isolated entity. Each pattern can exist in the world only to the extent that is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it.

—Christopher Alexander, architect and design theorist

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 Walley 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 and able to move 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.

One of the neat things about software design, certainly database design, is that there are base patterns, such as normalization, that we will build upon, but there are additional patterns that are built up starting with normalized tables. That is what this chapter is about, taking the basic structures we have built so far, and taking more and more complex groupings of structures, we will produce more complex, interesting solutions to problems.

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 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 these principals to produce the first manned 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. They weren't required to have their entire body scanned and patted down for that first flight, but the wings worked the same way.

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 is broken up into two major sections, starting

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_8

WOW! eBook
www.wowebook.org

with the larger topic, patterns that are desirable to use. The second major section discusses anti-patterns, or patterns that you may frequently see that are not desirable to use (along with the preferred method of solution, naturally).

Desirable Patterns

In this section, I am going to cover a variety of implementation patterns that can be used to solve 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 very common problems.

The patterns and solutions that I will present in the following subsections 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 to 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.
- *Historical/temporal*: At times it can be very desirable to look at previous versions of data that has changed over time. I will present strategies you can use to view your data at various points in history.
- *Hierarchies*: A very common need is to implement hierarchies in your data. The most common example is the manager-employee relationship. 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 user's avatar picture, 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*: We will look at some ways that you 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. 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 (`www.drsql.org`), I may make additional entries available over time, and please leave me comments if you have ideas for more.

WOW! eBook
www.wowebook.org

Uniqueness

If you have been reading this book straight through, you likely have gotten the point that uniqueness is a major concern for your design. 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 in some cases.

In this section, we will explore how you can implement different types of uniqueness issues that hit at the heart of the common 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, at least not recorded in the database.
- *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*: Instead of a single value of uniqueness, we often need 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. Things are even more important when you are controlling transportation systems.
- *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 purchase toy airplanes yesterday? Possibly. At the same phone number and address, with the same credit card type? Probably not. But another point I have worked to death is that in the database you can't enforce *probably*…only *definitely*.

Uniqueness is one of the biggest struggles in day-to-day operations, particularly in running a company, as it is essential to not offend customers, nor ship them 100 orders of Legos when they actually only placed a single order. We need to make sure that we don't end up with ten employees with the same SSN (and a visit from the tax man), far fewer cans of corn than we expected, ten appointments at the same time, and so on.

Selective Uniqueness

We previously discussed `PRIMARY KEY` and `UNIQUE` constraints, but neither of these will fit the scenario where you need to make sure some subset of the data, rather than every row, is unique. 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 three solutions to this problem that are common:

- *Filtered indexes*: This feature 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 versions prior to 2008, the way to implement this is to create a view that has a `WHERE` clause and then index the view.
- *Separate table for NULLable items*: A solution that needs noting is to make a separate table for the lower cardinality uniqueness items. So, for example, you might have a table for employees' insurance policy numbers. In some cases this may be the best solution, but it can lead to proliferation of tables that never really are accessed alone, making it more work than needed. If you designed properly, you will have decided on this in the design phase of the project.

WOW! eBook
www.wowebook.org

As a demonstration, I will create a schema and table for the human resources employee table with a column for employee number and a column for insurance policy number as well. I will use a database named Chapter8 with default settings for the examples unless otherwise noted.

```
CREATE SCHEMA HumanResources;
GO
CREATE TABLE HumanResources.Employee
(
    EmployeeId int IDENTITY(1,1) CONSTRAINT PKEmployee primary key,
    EmployeeNumber char(5) NOT NULL
           CONSTRAINT AKEmployee_EmployeeNummer UNIQUE,
    --skipping other columns you would likely have
    InsurancePolicyNumber char(10) NULL
);
```

Filtered indexes are useful for performance-tuning situations where only a few values are selective, but they also are useful for eliminating values for data protection. Everything about the index is the same as a normal index (indexes will be covered in greater detail in Chapter 10) save for the WHERE clause. So, you add an index like this:

```
--Filtered Alternate Key (AKF)
CREATE UNIQUE INDEX AKFEmployee_InsurancePolicyNumber ON
                                    HumanResources.Employee(InsurancePolicyNumber)
WHERE InsurancePolicyNumber IS NOT NULL;
```

Then, create an initial sample row:

```
INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES ('AO001','1111111111');
```

If you attempt to give another employee the same InsurancePolicyNumber

```
INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES ('AO002','1111111111');
```

this fails:

```
Msg 2601, Level 14, State 1, Line 29
Cannot insert duplicate key row in object 'HumanResources.employee' with unique index
'AKFEmployee_InsurancePolicyNumber'. The duplicate key value is (1111111111).
```

Adding the row with the corrected value will succeed:

```
INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES ('AO002','2222222222');
```

However, adding two rows with NULL will work fine:

```
INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES ('AO003','3333333333'),
```


WOW! eBook
www.wowebook.org

```
        ('A0004',NULL),
        ('A0005',NULL);
```

You can see that this

```
SELECT *
FROM   HumanResources.Employee;
```

returns the following:

```
EmployeeId       EmployeeNumber          InsurancePolicyNumber
---------------- ----------------------- --------------------------------
1                A0001                   1111111111
3                A0002                   2222222222
4                A0003                   3333333333
5                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
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 PKContact 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 AKFContact_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
VALUES ('bob','11111',1);
GO
INSERT INTO Account.Contact
VALUES ('fred','11111',1);
```


WOW! eBook
www.wowebook.org

the following error is returned after the second insert:

```
Msg 2601, Level 14, State 1, Line 73
Cannot insert duplicate key row in object 'Account.Contact' with unique index 'AKFContact_
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
VALUES ('fred','11111', 1);

COMMIT TRANSACTION;
```

Note that in cases like this you would definitely want to use a transaction and error handling in your code so you don't end up without a primary contact if the `INSERT` operation 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 with a unique clustered indexes. 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 if you cannot use a filtered index (though as of this writing, it is available in all supported versions of SQL Server.

A side effect of the filtered index is that it (like the uniqueness constraints we have used previously) has a very good chance of being useful for searches against the table. The only downside is that the error comes from an index rather than a constraint, so it does not fit into our existing paradigms for error handling.

Bulk Uniqueness

Sometimes, we need to inventory items where some of the items are equivalent in the physical world, for example, cans of corn in the grocery store. Generally, you can't even tell the cans apart by looking at them (unless they have different expiration dates, perhaps), but knowing how many are in stock is a very common need. 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. 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. Figure 8-1 shows a very simplified model of such activity.

WOW! eBook
www.wowebook.org

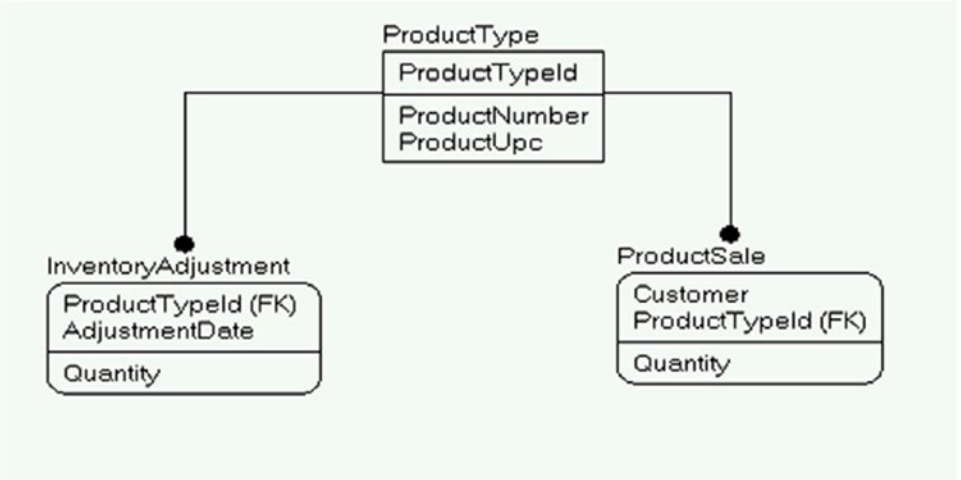


Figure 8-1. *Simplified inventory model*

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 so forth, and in the `ProductSale` table (probably a child to sale header or perhaps invoicing table in a complete model), you would record when product is removed or added to inventory from a customer interaction.

The sum of the `InventoryAdjustment Quantity` value less the sum of `ProductSale Quantity` value should tell you the amount of product on hand (or perhaps the amount of product you have oversold and need to order posthaste!) In the more realistic case, you would have a lot of complexity for backorders, future orders, returns, and so on, but the base concept is basically the same. Instead of each row representing a single item, each 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.

WOW! eBook
www.wowebook.org

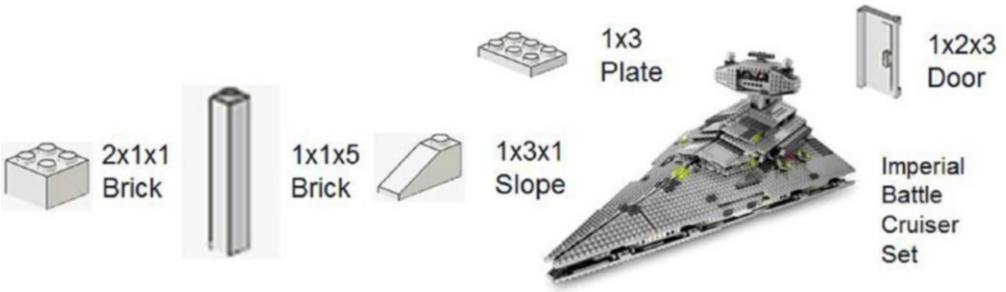


Figure 8-2. *Sample Lego parts for a database*

To solve this problem, I will create a table for each set of Legos I own (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 CONSTRAINT PKBuild PRIMARY KEY,
        Name    varchar(30) NOT NULL CONSTRAINT AKBuild_Name UNIQUE,
        LegoCode varchar(5) NULL, --five character set number
        InstructionsURL varchar(255) NULL --where you can get the PDF of the instructions
);
```

Then, I'll add a table for each individual instance of that build, which I will call `BuildInstance`:

```
CREATE TABLE Lego.BuildInstance
(
        BuildInstanceId Int CONSTRAINT PKBuildInstance PRIMARY KEY ,
        BuildId Int CONSTRAINT FKBuildInstance$isAVersionOf$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 AKBuildInstance 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 PKPiece 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,
```


WOW! eBook
www.wowebook.org

```
        LegoInventoryNumber int NULL,
        OwnedCount int NOT NULL,
        CONSTRAINT AKPiece_Definition UNIQUE (Type,Name,Color,Width,Length,Height),
        CONSTRAINT AKPiece_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. The likely use here will be to update the value as new pieces are added to inventory and possibly to count up loose pieces later and add that value to the ones in sets (which we will have a query for later).

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 PKBuildInstancePiece PRIMARY KEY (BuildInstanceId, PieceId)
);
```

From here, I can load some data. I will load a true Lego item that is available for sale 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 personally have only 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),
```


WOW! eBook
www.wowebook.org

```
(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),
(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 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);
```

Next, 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
```


WOW! eBook
www.wowebook.org

```
INSERT INTO Lego.BuildInstancePiece (BuildInstanceId, PieceId, AssignedCount)
VALUES (3,1,3);
```

After the mundane (and quite tedious when done all at once) business of setting up the data 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;
```

This query returns the following, with the first column giving us the different types:

```
PieceCount  InventoryCount
----------- --------------
30          111
```

Here, you start to get a feel for how this is going to be a different sort of solution than the basic relational inventory solution. Instinctively, one expects that a single 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 based 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 2 types of brick but 30 bricks in inventory, 1 type of baseplate but 4 of them in inventory, 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
```


WOW! eBook
www.wowebook.org

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 1 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 `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 wonderful feature that comes in handy 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
                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
```


WOW! eBook
www.wowebook.org

```
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:

```
;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 almost 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).

WOW! eBook
www.wowebook.org

Range Uniqueness

In some cases, uniqueness isn't uniqueness on the values of a single column set, but rather over the values between 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.

For example, consider an appointment time. It has a start and an end, and the start and end ranges of two appointments should not overlap. Suppose we have an appointment with start and end times defined with precision to the second, starting at `'20160712 1:00:00PM'` and ending at `'20160712 1:59:59PM'`. To validate that this data does not overlap other appointments, we need to look for rows where any of the following conditions are met, indicating we are double booking appointment times:

- 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.

We can protect against situations such as overlapping appointment times by employing a trigger and a query that checks for range overlapping. If the aforementioned 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 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:

```
CREATE SCHEMA Office;
GO
CREATE TABLE Office.Doctor
(
        DoctorId        int NOT NULL CONSTRAINT PKDoctor PRIMARY KEY,
        DoctorNumber char(5) NOT NULL CONSTRAINT AKDoctor_DoctorNumber UNIQUE
);
CREATE TABLE Office.Appointment
(
        AppointmentId   int NOT NULL CONSTRAINT PKAppointment PRIMARY KEY,
        --real situation would include room, patient, etc,
        DoctorId        int NOT NULL,
        StartTime       datetime2(0), --precision to the second
        EndTime         datetime2(0),
        CONSTRAINT AKAppointment_DoctorStartTime UNIQUE (DoctorId,StartTime),
        CONSTRAINT AKAppointment_DoctorEndTime UNIQUE (DoctorId,EndTime),
        CONSTRAINT CHKAppointment_StartBeforeEnd CHECK (StartTime <= EndTime),
        CONSTRAINT FKDoctor$IsAssignedTo$OfficeAppointment FOREIGN KEY (DoctorId)
                                        REFERENCES Office.Doctor (DoctorId)
);
```

Next, we will add some data to our new table. The `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,'20160712 14:00','20160712 14:59:59'),
       (2,1,'20160712 15:00','20160712 16:59:59'),
```


WOW! eBook
www.wowebook.org

```
        (3,2,'20160712 8:00','20160712 11:59:59'),
        (4,2,'20160712 13:00','20160712 17:59:59'),
        (5,2,'20160712 14:00','20160712 14:59:59'); --offensive item for demo, conflicts
                                                    --with 4
```

As far as the declarative constraints can tell, everything is okay, but the following query will check for data conditions between each row in the table to every other row in the table:

```
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. 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 an other.

Running the query, we see that

```
AppointmentId ConflictingAppointmentId
------------- ------------------------
5             4
4             5
```

The interesting part of these results is that where there is one offending row, there will always be another. If one row is offending in one way, like starting before and ending after another appointment, the conflicting row will have a start and end time between the first appointment's time. This won't be a problem, but the shared blame makes 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) that will check for this condition based on the values in new rows being inserted or updated. There's no need to check deleted rows (even for the update case), because all a delete operation can do is help the situation (even in the case of the update, where you may move an appointment from one doctor to another).

WOW! eBook
www.wowebook.org

Note that the basis of this trigger is the query we used previously to check for bad values (I usually implement this as two triggers, one for insert and another for update, both having the same code, but it is shown here as one for simplicity of demonstration):

```
CREATE TRIGGER Office.Appointment$insertAndUpdate
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 this is an update, but they don't change times or doctor, don't check the data
        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
                                          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 ' +
```


WOW! eBook
www.wowebook.org

```
                                                    '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;
```

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           2016-07-12 14:00:00   2016-07-12 14:59:59
2             1           2016-07-12 15:00:00   2016-07-12 16:59:59
3             2           2016-07-12 08:00:00   2016-07-12 11:59:59
4             2           2016-07-12 13:00:00   2016-07-12 17:59:59
```

This time, when we try to add an appointment for `doctorId` number 1:

```
INSERT INTO Office.Appointment
VALUES (5,1,'20160712 14:00','20160712 14:59:59');
```

this first attempt is blocked because the row is an exact duplicate of the start time value. The most common error that will likely occur in a system such as this is trying to duplicate something, usually by accident.

```
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, 2016-07-12
14:00:00).
```

Next, we check the case where the appointment fits wholly inside of another appointment:

```
INSERT INTO Office.Appointment
VALUES (5,1,'20160712 14:30','20160712 14:40:59');
```


WOW! eBook
www.wowebook.org

This fails and tells us the doctor for whom the failure occurred:

```
Msg 50000, Level 16, State 16, Procedure appointment$insertAndUpdate, Line 48
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,'20160712 11:30','20160712 17:59:59');
```

This quite obediently fails, just like the other case:

```
Msg 50000, Level 16, State 16, Procedure appointment$insertAndUpdate, Line 48
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:

```
INSERT into Office.Appointment
VALUES (5,1,'20160712 11:30','20160712 15:59:59'),
       (6,2,'20160713 10:00','20160713 10:59:59');
```

This time, it fails with our multirow error message:

```
Msg 50000, Level 16, State 16, Procedure appointment$insertAndUpdate, Line 48
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,'20160712 10:00','20160712 11:59:59'),
       (6,2,'20160713 10:00','20160713 10:59:59');
```

This will (finally) work. Now, test failing an update operation:

```
UPDATE Office.Appointment
SET    StartTime = '20160712 15:30',
       EndTime = '20160712 15:59:59'
WHERE  AppointmentId = 1;
```

This fails like it should:

```
Msg 50000, Level 16, State 16, Procedure appointment$insertAndUpdate, Line 38
Appointment for doctor 00001 overlapped existing appointment
```


WOW! eBook
www.wowebook.org

If this seems like a lot of work, it 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.

Instead of blocking the operation, the trigger could update rows to tell the user that it overlaps. Or you might even get more interesting and let the algorithms involve prioritizing certain conditions above other appointments. Maybe a checkup gets bumped for a surgery, marked to reschedule. Realistically, it may be that the user can overlap appointments all they want, and then at the close of business, a query such as the one that formed the basis of the trigger is executed by the administrative assistant, who then clears up any scheduling issues manually. In this section, I've shown you a pattern to apply to prevent range overlaps if that is the desired result. It is up to the requirements to lead you to exactly how to implement.

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 Abraham Lincoln, are they the same person or are they two aliases someone is using? (Or one alias and one real name?) 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 Abraham Lincoln the data of the real Abraham Lincoln, well, that just isn't going to be good no matter what your privacy policy is or which laws that govern privacy apply). 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
- Changing faster than a printed book can cover
- Different depending on what part of the world you are reading this book in

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. (Chapter 9 will expand a little bit on this topic as we talk about encryption technologies, but encryption is another wide topic for which the best advice is to make sure you are doing as much or more as is required.)

So the goal of your design is to work at getting your customer to use an identifier to help you distinguish them from another customer. This customer identifier will be used, for example, 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 be gathered from a web browser, telephone number, and so on. Even worse, what if someone signs up twice for a customer number? Can you be sure that it is the same person, then?

WOW! eBook
www.wowebook.org

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)

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 customer 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 get in the habit of doing is programming using specific values, to force a specific action. For example, they will get requirements that specify that for customers 1 and 2, we need to do action A, and for customer 3, we need to do action B. So they go in and code the following:

```
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 a month later, they add `'4'` to the code, test it, deploy it, and claim it required 40 hours of IT 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` column 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:

WOW! eBook
www.wowebook.org

```
--In real table, expand ActionType to be a more descriptive value or a domain of its own
CREATE SCHEMA Customers;
GO
CREATE TABLE Customers.CustomerType
(
        CustomerType    varchar(20) NOT NULL CONSTRAINT PKCustomerType PRIMARY KEY,
        Description     varchar(1000) NOT NULL,
        ActionType      char(1) NOT NULL CONSTRAINT CHKCustomerType_ActionType_Domain
                                        CHECK (ActionType 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. 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.

Historical/Temporal Data

One pattern that is quite often needed is to be able to see how a row looked at a previous point in time. For example, when did Employee 100001's salary change? When did Employee 2010032's insurance start? In some cases, you need to capture changes using a column in a table. For example, take the Employee table we used in the "Selective Uniqueness" section (if you are following along, this table already exists in the database, so you don't need to re-create it):

```
CREATE TABLE HumanResources.Employee
(
    EmployeeId int IDENTITY(1,1) CONSTRAINT PKEmployee primary key,
    EmployeeNumber char(5) NOT NULL
           CONSTRAINT AKEmployee_EmployeeNummer UNIQUE,
    InsurancePolicyNumber char(10) NULL
);
CREATE UNIQUE INDEX AKFEmployee_InsurancePolicyNumber ON
                                    HumanResources.Employee(InsurancePolicyNumber)
                                    WHERE InsurancePolicyNumber IS NOT NULL;
```

One method of answering the question of when the insurance number changed would be to add

```
ALTER TABLE HumanResources.Employee
     ADD InsurancePolicyNumberChangeTime datetime2(0);
```


WOW! eBook
www.wowebook.org

Possibly with the addition of who changed the row. It could be managed with a trigger, or from the interface code. If you are dealing with a single column that needs to be verified, this is a great way to handle this need. But if you really want to see how all of the columns of a table have changed over time (or a larger subset than one for sure), there are two techniques using T-SQL that are quite popular:

- *Using a trigger*: Using a fairly simple trigger, we can capture changes to the row and save them in a separate table. The benefit of this method over the next method is that it does not need to capture change for all columns.
- *Using temporal extensions*: These are available in SQL Server 2016 and allow you to let the engine capture the changes to a table. The major benefit is that there is syntax that allows you to query the changed data in a quite simple manner.

In this section I will demonstrate both methods using the table we have created. Before starting each example, I will use the following code to reset the tables of data:

```
TRUNCATE TABLE HumanResources.Employee;

INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES ('A0001','1111111111'),
       ('A0002','2222222222'),
       ('A0003','3333333333'),
       ('A0004',NULL),
       ('A0005',NULL),
       ('A0006',NULL);
```

Note that neither version of this pattern is generally a version of auditing. The trigger model can be used more for auditing by adding columns in for who made the change, but there are better auditing tools built in to find people doing incorrect things with your data. This pattern is specifically set up to show changes in data, and it is not beyond reasonability that you might change history to cover up mistakes that were data related. Audits should *never* be changed or they will quickly be considered unreliable.

Using a Trigger to Capture History

While temporal support is a big new feature of SQL Server 2016, it does not take away the value of using a trigger to capture history. In this section, our goal is simply to see a log of modified rows, or rows that have been deleted from the table. For this exercise, we will start by creating a schema for the history that corresponds to the name of the schema that owns the data. This will allow you to manage security at the schema level for history different than for the base table. So giving the user `SELECT` rights of a schema to read the data does not give them rights to see the history data.

```
CREATE SCHEMA HumanResourcesHistory;
```

Next we will create a parallel history table in the new schema that has all of the columns of the original table, along with a few management columns, which have explanations in the following code:

```
CREATE TABLE HumanResourcesHistory.Employee
(
    --Original columns
    EmployeeId int NOT NULL,
```


WOW! eBook
www.wowebook.org

```
    EmployeeNumber char(5) NOT NULL,
    InsurancePolicyNumber char(10) NULL,

    --WHEN the row was modified
    RowModificationTime datetime2(7) NOT NULL,
    --WHAT type of modification
    RowModificationType varchar(10) NOT NULL CONSTRAINT
              CHKEmployeeSalary_RowModificationType
                    CHECK (RowModificationType IN ('UPDATE','DELETE')),
    --tiebreaker for seeing order of changes, if rows were modified rapidly
    RowSequencerValue bigint IDENTITY(1,1) --use to break ties in RowModificationTime
);
```

Next, we create the following trigger. The basic flow is to determine the type of operation, then write the contents of the deleted table to the history table we have previously created.

```
CREATE TRIGGER HumanResources.Employee$HistoryManagementTrigger
ON HumanResources.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
           @rowsAffected int = (SELECT COUNT(*) FROM deleted);
   IF @rowsAffected = 0 RETURN;

   DECLARE @RowModificationType char(6);
   SET @RowModificationType = CASE WHEN EXISTS (SELECT * FROM inserted) THEN 'UPDATE'
                                                                  ELSE 'DELETE' END;
   BEGIN TRY
       --[validation section]
       --[modification section]
       --write deleted rows to the history table
       INSERT  HumanResourcesHistory.Employee(EmployeeId,EmployeeNumber,InsurancePolicyNumber,
                                              RowModificationTime,RowModificationType)
       SELECT EmployeeId,EmployeeNumber,InsurancePolicyNumber,
              SYSDATETIME(), @RowModificationType
       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;
```


WOW! eBook
www.wowebook.org

Now let's make a few changes to the data in the table. As a reminder, here is the data we start with:

```
SELECT *
FROM   HumanResources.Employee;
```

This shows us our base data:

```
----------- -------------- ---------------------
1           A0001          1111111111
2           A0002          2222222222
3           A0003          3333333333
4           A0004          NULL
5           A0005          NULL
6           A0006          NULL
```

Next, update `EmployeeId = 4` and set that it has insurance:

```
UPDATE HumanResources.Employee
SET    InsurancePolicyNumber = '4444444444'
WHERE  EmployeeId = 4;
```

You can see the change and the history here:

```
SELECT *
FROM   HumanResources.Employee
WHERE  EmployeeId = 4;
SELECT *
FROM   HumanResourcesHistory.Employee
WHERE  EmployeeId = 4;
EmployeeId  EmployeeNumber InsurancePolicyNumber
----------- -------------- ---------------------
4           A0004          4444444444

EmployeeId  EmployeeNumber InsurancePolicyNumber RowModificationTime
----------- -------------- --------------------- --------------------------
4           A0004          NULL                  2016-05-07 20:26:38.4578351

RowModificationType RowSequencerValue
------------------- --------------------
UPDATE              1
```

Now let's update all of the rows where there is an insurance policy to a new format, and delete `EmployeeId = 6`. I updated all rows so we can see in the history what happens when a row is updated and does not actually change:

```
UPDATE HumanResources.Employee
SET  InsurancePolicyNumber = 'IN' + RIGHT(InsurancePolicyNumber,8);

DELETE HumanResources.Employee
WHERE EmployeeId = 6;
```


WOW! eBook
www.wowebook.org

Then check out the data:

```
SELECT *
FROM   HumanResources.Employee
ORDER BY EmployeeId;

--limiting output for formatting purposes
SELECT EmployeeId, InsurancePolicyNumber, RowModificationTime, RowModificationType
FROM   HumanResourcesHistory.Employee
ORDER BY EmployeeId,RowModificationTime,RowSequencerValue;
```

This returns

```
EmployeeId  EmployeeNumber InsurancePolicyNumber
----------- -------------- --------------------
1           A0001          IN11111111
2           A0002          IN22222222
3           A0003          IN33333333
4           A0004          IN44444444
5           A0005          NULL

EmployeeId   InsurancePolicyNumber RowModificationTime         RowModificationType
-----------  -------------------- --------------------------- -------------------
1            1111111111           2016-05-07 20:27:59.8852810 UPDATE
2            2222222222           2016-05-07 20:27:59.8852810 UPDATE
3            3333333333           2016-05-07 20:27:59.8852810 UPDATE
4            NULL                 2016-05-07 20:26:38.4578351 UPDATE
4            4444444444           2016-05-07 20:27:59.8852810 UPDATE
5            NULL                 2016-05-07 20:27:59.8852810 UPDATE
6            NULL                 2016-05-07 20:27:59.8852810 UPDATE
6            NULL                 2016-05-07 20:27:59.9347658 DELETE
```

Using this data, you can see the progression of what has happened to the data, starting from the time the trigger was added, including seeing rows that have been deleted. It is a well-worn method to capture history, but it is very hard to work with in your queries to look back on history.

Using Temporal Extensions to Manage History

Temporal extensions will provide the same basic information that we provided using the trigger in the previous section, but there is one major difference: query support. If you want to see the current data, there is no change to your query. But if you want to see how the data looked at a particular point in time, the only change to your query is to specify the time (or time range) for which you want to see history.

The biggest limitation on temporal tables is that the history copy must version all columns in the table. So if you are using an `nvarchar(max)` or even `text` (which you really shouldn't be!), it will work, but you could incur massive performance issues if your values are very large. Even in-memory tables support temporal, but version table will be an on-disk table.

There are other limitations, such as requiring a primary key; `TRUNCATE TABLE` not allowed; foreign key `CASCADE` operations are not allowed on the table; `INSTEAD OF` triggers are not allowed on the table; and replication use is limited. Several other configuration limitations are included in this more complete list of considerations and limitations from Microsoft: msdn.microsoft.com/en-us/library/mt604468.aspx.

WOW! eBook
www.wowebook.org

However, the limitations are not terribly constraining if you need the DML extensions described later in the section.

I will continue to use the HumanResources table we used in the previous section, but I will reset the data and drop the history table from the previous section:

```
TRUNCATE TABLE HumanResources.Employee;

INSERT INTO HumanResources.Employee (EmployeeNumber, InsurancePolicyNumber)
VALUES ('A0001','1111111111'),
       ('A0002','2222222222'),
       ('A0003','3333333333'),
       ('A0004',NULL),
       ('A0005',NULL),
       ('A0006',NULL);
GO
DROP TABLE HumanResourcesHistory.Employee;
DROP TRIGGER HumanResources.Employee$HistoryManagementTrigger;
```

In the following subsections, I will cover configuring temporal extensions, along with how you can coordinate changes to multiple rows and how you can change history if needed.

Configuring Temporal Extensions

Now let's add temporal extensions to the HumanResources.Employee table. To do this, you are required to have two columns in the table, one for a start time, which you will see often in the documentation as SysStartTime, and one for an end time, usually named SysEndTime. I will use a naming standard that matches my normal standard so they match the table. These time range columns must be datetime2, but they can be any precision from 0 to 7. These values are used to denote the start and end time that the row is valid, so when we query the rows using temporal extensions, the one active row can be picked. The precision of the column will determine how many versions you can have per second. If datetime2(0) is used, then you can have one version per second; if datetime2(7), then ~9999999 versions per second. While it is not likely that many readers will have such needs to track changes to this deep level, I tend to use (7) just because it feels like it is safer, if more difficult to type. (For this book I will use datetime2(1) to allow for the limited amount of text real estate I have available.)

The following code snippet adds the columns and the settings that we will use once we turn on system versioning (then drops the temporary constraints):

```
ALTER TABLE HumanResources.Employee
ADD
    RowStartTime datetime2(1) GENERATED ALWAYS AS ROW START NOT NULL
         --HIDDEN can be specified
         --so temporal columns don't show up in SELECT * queries
         --This default will start the history of all existing rows at the
         --current time (system uses UTC time for these values)
        CONSTRAINT DFLTDelete1 DEFAULT (SYSUTCDATETIME()),
    RowEndTime datetime2(1) GENERATED ALWAYS AS ROW END NOT NULL --HIDDEN
          --data needs to be the max for the datatype
        CONSTRAINT DFLTDelete2 DEFAULT (CAST('9999-12-31 23:59:59.9' AS datetime2(1)))
  , PERIOD FOR SYSTEM_TIME (RowStartTime, RowEndTime);
```


WOW! eBook
www.wowebook.org

```
GO

--DROP the constraints that are just there due to data being in the table
ALTER TABLE HumanResources.Employee
        DROP CONSTRAINT DFLTDelete1;
ALTER TABLE HumanResources.Employee
         DROP CONSTRAINT DFLTDelete2;
```

The GENERATED ALWAYS AS ROW START and END pair tells the system to set the value when the table is completely configured for temporal support. If you are creating a new table and want to turn on temporal extensions, you will use the same columns and settings in the CREATE TABLE statement, but you won't need the DEFAULT constraints.

The next step is to create a version table. There are two ways to do this. The easiest is to just let SQL Server build it for you. You can either specify a name or let SQL Server pick one for you. For example, if we want SQL Server to create the history table, we will just use

```
ALTER TABLE HumanResources.Employee
         SET (SYSTEM_VERSIONING = ON);
```

Now you can look in the system metadata and see what has been added:

```
SELECT  tables.object_id AS baseTableObject,
        CONCAT(historySchema.name,'.',historyTable.name) AS historyTable
FROM    sys.tables
          JOIN sys.schemas
              ON schemas.schema_id = tables.schema_id
          LEFT OUTER JOIN sys.tables AS historyTable
                 JOIN sys.schemas AS historySchema
                       ON historySchema.schema_id = historyTable.schema_id
            ON TABLES.history_table_id = historyTable.object_id
WHERE   schemas.name = 'HumanResources'
  AND   tables.name = 'Employee';
```

This returns something like the following, with almost certainly a different base table object_id:

```
baseTableObject historyTable
--------------- ------------------------------------------------------------
1330103779      HumanResources.MSSQL_TemporalHistoryFor_1330103779
```

This leads to a predictable but ugly name. The table will have the same columns as the base table, but will have a few differences that we will look at later when we cover creating your own table and modifying the data in the table to use previous historical data you have saved off.

While there may not be a reason all that often to look at the temporal tables, it will be useful to be able to correlate the names of the tables without knowing the object_id. So let's go ahead and name the table ourselves in the DDL. First we need to disconnect the history table that was created and drop it. Before you can do much to the base table, in fact, you will have to turn off the system versioning:

```
ALTER TABLE HumanResources.Employee
         SET (SYSTEM_VERSIONING = OFF);
DROP TABLE HumanResources.MSSQL_TemporalHistoryFor_1330103779;
```


WOW! eBook
www.wowebook.org

Now let's specify the table name. If it is an existing table, there will be more to do, in that you may want to backfill up history (like if you were previously trigger to capture history). Just like in the trigger method, I will use a different schema for the history tables, but you can put it in the same schema if so desired (you always have to specify the schema in the HISTORY_TABLE clause):

```
ALTER TABLE HumanResources.Employee
                                    --must be in the same database
        SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE = HumanResourcesHistory.Employee));
```

Taking a look at the metadata query we ran earlier, you can see that the history table has been set to a much better table name:

```
baseTableObject historyTable
--------------- -------------------------------------
1330103779      HumanResourcesHistory.Employee
```

I will cover security in more detail in Chapter 9, but understand that the user will need rights to the history table in order to SELECT the temporal aspects of the table. A user can modify the contents of the table with just INSERT, UPDATE, and/or DELETE rights.

Now you have configured the HumanResources.Employee table to capture history, starting with the time period of your ALTER statement to add the columns to the table. Check the table's content using a SELECT statement:

```
SELECT *
FROM   HumanResources.Employee;
```

This returns

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
1           A0001          1111111111            2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
2           A0002          2222222222            2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
3           A0003          3333333333            2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
4           A0004          NULL                  2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
5           A0005          NULL                  2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
6           A0006          NULL                  2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
```

You can see the new columns added for RowStart… and RowEnd… Time. Using these timeframes, you will be able to see the data at given points of time. So if you wanted to see how the table would have looked on the fourth of May, use the FOR SYSTEM_TIME clause on the table in the FROM clause using AS OF a current point in time, in our case where RowStartTime >= PassedValue > RowEndTime. There are four others: FROM, BETWEEN, CONTAINED IN, and ALL. I will mostly make use of AS OF and ALL in the book, as usually I want to see data at a point in time, or I want to see all history to show you what has changed.

```
SELECT *
FROM   HumanResources.Employee FOR SYSTEM_TIME AS OF '2016-05-04';
```


WOW! eBook
www.wowebook.org

This returns nothing, as the RowStart and End do not include that time period for any row in the table. The following query will (based on the data as I have it in my sample table) return the same as the previous query to get all rows in the base table, since 2016-05-11 is after all of the RowStartTime values in the base table:

```
SELECT *
FROM   HumanResources.Employee FOR SYSTEM_TIME AS OF '2016-05-11';
```

Dealing with Temporal Data One Row at a Time

When your application modifies a single row in a table that has temporal extensions enabled, there really isn't much you need to consider in your application. Every INSERT, UPDATE, and DELETE operation will just capture changes, and let you query each table that is involved in the operation at a point in time. You can use the FROM FOR SYSTEM_TIME clause on any statement where a FROM clause makes sense. And you can use it on all tables that are used in a query, or just some. For example, the following is perfectly acceptable:

```
FROM  Table1
        JOIN Table2 FOR SYSTEM_TIME AS OF 'Time Literal'
                    ON ...
        JOIN Table3 FOR SYSTEM_TIME AS OF 'Time Literal'
                    ON ...
```

And you can even do this:

```
FROM  Table1 FOR SYSTEM_TIME AS OF 'Time Literal 1'
        JOIN Table1 as DifferentLookAtTable1 FOR SYSTEM_TIME AS OF 'Time Literal 2'
                    ON ...
```

In the next section we will look more at coordinating modifications on multiple rows (in the same or multiple tables), but in this section, let's look at the basic mechanics.

First let's modify some data, to show what this looks like:

```
UPDATE HumanResources.Employee
SET    InsurancePolicyNumber = '4444444444'
WHERE  EmployeeId = 4;
```

So let's look at the data:

```
SELECT *
FROM   HumanResources.Employee
WHERE  EmployeeId = 4;
```

As expected:

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
4           A0004          4444444444            2016-05-10 02:46:58.3 9999-12-31 23:59:59.9
```


WOW! eBook
www.wowebook.org

But check just before the RowStartTime (.3 seconds to be precise):

```
SELECT *
FROM   HumanResources.Employee FOR SYSTEM_TIME AS OF '2016-05-10 02:46:58'
WHERE  EmployeeId = 4;
```

and the data looks just the same as it did pre-UPDATE execution:

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
4           A0004          NULL                  2016-05-10 02:35:49.1 2016-05-10 02:46:58.3
```

This is where ALL comes in handy, so you can see all of the changes:

```
SELECT *
FROM   HumanResources.Employee FOR SYSTEM_TIME ALL
ORDER  BY EmployeeId, RowStartTime;
```

This returns all of the valid history rows (ones where RowStartTime <> RowEndTime, a situation that I will cover in a moment), including previous versions of data:

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
1           A0001          1111111111            2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
2           A0002          2222222222            2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
3           A0003          3333333333            2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
4           A0004          NULL                  2016-05-10 02:35:49.1 2016-05-10 02:46:58.3
4           A0004          4444444444            2016-05-10 02:46:58.3 9999-12-31 23:59:59.9
5           A0005          NULL                  2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
6           A0006          NULL                  2016-05-10 02:35:49.1 9999-12-31 23:59:59.9
```

Now, let's delete EmployeeId = 6 (we don't like him…he ate our gluten-free, 10-calorie cupcake):

```
DELETE HumanResources.Employee
WHERE  EmployeeId = 6;
```

Then check out the data:

```
SELECT *
FROM   HumanResources.Employee FOR SYSTEM_TIME ALL
WHERE  EmployeeId = 6
ORDER  BY EmployeeId, RowStartTime;
```

Now you can see that the RowEndTime value is not '9999-12-31 23:59:59.9' but is set to the time of the DELETE:

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
6           A0006          NULL                  2016-05-10 02:35:49.1 2016-05-10 05:36:28.3
```


WOW! eBook
www.wowebook.org

The reasons is that at that point in time, it *did* exist, but now it doesn't. If he (`EmployeeId` = 6) apologizes and gets added back with the same surrogate key value, there would be a gap in time sequence that would correspond to the time when the row was removed.

A word of caution about versions. When the table has `SYSTEM_VERSIONING ON`, every update will cause a new version, even if no data changes. So,

```
UPDATE HumanResources.Employee
SET    EmployeeNumber = EmployeeNumber
WHERE  EmployeeId = 4;
```

is going to lead to a new version:

```
SELECT *
FROM   HumanResources.Employee FOR SYSTEM_TIME ALL
WHERE  EmployeeId = 4
ORDER  BY EmployeeId, RowStartTime;
```

This returns

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
4           A0004          NULL                  2016-05-10 02:35:49.1 2016-05-10 02:46:58.3
4           A0004          4444444444            2016-05-10 02:46:58.3 2016-05-10 02:54:11.8
4           A0004          4444444444            2016-05-10 02:54:11.8 2016-05-10 02:59:36.6
```

But you can see that there is no difference between the second row in the output and the third. Now, let's do five updates, immediately following one another using SSMS's `GO #` extension:

```
UPDATE HumanResources.Employee
SET    EmployeeNumber = EmployeeNumber
WHERE  EmployeeId = 4;
GO 5
```

Looking at the data

```
SELECT *
FROM   HumanResources.Employee FOR SYSTEM_TIME ALL
WHERE  EmployeeId = 4
ORDER  BY EmployeeId, RowStartTime;
```

we see only five rows, but there should be eight, right? In this case, some rows were updated within what the system registered as the same time, so they are hidden. You can only see them in the history table:

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
4           A0004          NULL                  2016-05-10 02:35:49.1 2016-05-10 02:46:58.3
4           A0004          4444444444            2016-05-10 02:46:58.3 2016-05-10 02:54:11.8
4           A0004          4444444444            2016-05-10 02:54:11.8 2016-05-10 02:59:36.6
4           A0004          4444444444            2016-05-10 02:59:36.6 2016-05-10 03:01:06.5
4           A0004          4444444444            2016-05-10 03:01:06.5 9999-12-31 23:59:59.9
```


WOW! eBook
www.wowebook.org

Three of the rows have the same RowStart and RowEnd times (your mileage may vary, depending on the precision of your row times; sometimes I did not see this phenomenon, but most every time I did, even using datetime2(7)):

```
SELECT *
FROM   HumanResourcesHistory.Employee
WHERE  EmployeeId = 4
  AND  RowStartTime = RowEndTime;
```

Here you see the remaining three rows:

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
4           A0004          4444444444            2016-05-10 03:01:06.5 2016-05-10 03:01:06.5
4           A0004          4444444444            2016-05-10 03:01:06.5 2016-05-10 03:01:06.5
4           A0004          4444444444            2016-05-10 03:01:06.5 2016-05-10 03:01:06.5
```

If your application is very chatty and updates the same row over and over, you could end up with a lot of useless version rows.

■ **Tip** Another way you end up with start and end times that are the same is when you modify the same row multiple times in the same transaction. All of these modifications are hidden to the FOR SYSTEM_TIME, but can be seen in the history table.

Now that you have started accumulating history, you are free to query your data at any point in time, down to whatever your precision is set to. How did the data look yesterday at this point in time? Or the previous day at 11:00 AM? Compared to now? The value of this could be enormous. However, as we will dig deeper into in the next section, it brings up a problem. Now you can't limit your thinking to just one point in time. Every time slice should be synchronized.

What if you accidentally set the InsurancePolicyNumber to NULL? Or you set it to an incorrect value? In a regular situation, you update the row and all is great. But if you are using your temporal versions to look back at your database at a point in time, reports may not look correct. You cannot simply update the history table, but rather have to turn off versioning, fix the history, and turn versioning back on. I will cover the process in the forthcoming section "Setting/Rewriting History," but it is something to be done afterhours when no one can access the table, which is not optimal.

Dealing with Multiple Rows in One or More Tables

At the end of the previous section, I started the discussion about thinking temporally with one row at a time. Versions throughout history need coordinated data that does not tell any falsehoods (even if you have cleared them up later in your base table). In this section we will extend the concept to multiple rows. If you change two rows in a table, the only easy way to make sure their historical time values are the same if you are building your own using triggers is to make sure you do your update in a single statement. The temporal extensions give you a much better method of synchronizing changes. Basically, the start and end time columns are set at COMMIT time in the transaction. If you want to update all of the InsurancePolicyNumber values to include the letters 'IN' as a prefix, and for some reason, you were unable to do this in a single statement (not every example can be realistic!), you wrap the change into a BEGIN and COMMIT transaction:

WOW! eBook
www.wowebook.org

```
BEGIN TRANSACTION;

UPDATE HumanResources.Employee
SET    InsurancePolicyNumber = CONCAT('IN',RIGHT(InsurancePolicyNumber,8))
WHERE  EmployeeId = 1;

WAITFOR DELAY '00:00:01';

UPDATE HumanResources.Employee
SET    InsurancePolicyNumber = CONCAT('IN',RIGHT(InsurancePolicyNumber,8))
WHERE  EmployeeId = 2;

WAITFOR DELAY '00:00:01';

UPDATE HumanResources.Employee
SET    InsurancePolicyNumber = CONCAT('IN',RIGHT(InsurancePolicyNumber,8))
WHERE  EmployeeId = 3;

WAITFOR DELAY '00:00:01';

UPDATE HumanResources.Employee
SET    InsurancePolicyNumber = CONCAT('IN',RIGHT(InsurancePolicyNumber,8))
WHERE  EmployeeId = 4;

COMMIT TRANSACTION;
```

Looking at the data

```
SELECT *
FROM   HumanResources.Employee
WHERE  InsurancePolicyNumber IS NOT NULL
ORDER BY EmployeeId;
```

you can see that the `RowStartTime` values for every row that was updated is exactly the same, even if the `UPDATE` statements weren't actually executed at the same point in time:

EmployeeId	EmployeeNumber	InsurancePolicyNumber	RowStartTime	RowEndTime
1	A0001	IN11111111	2016-05-10 03:08:50.9	9999-12-31 23:59:59.9
2	A0002	IN22222222	2016-05-10 03:08:50.9	9999-12-31 23:59:59.9
3	A0003	IN33333333	2016-05-10 03:08:50.9	9999-12-31 23:59:59.9
4	A0004	IN44444444	2016-05-10 03:08:50.9	9999-12-31 23:59:59.9

For simplicity's sake, I won't try to show multiple tables as an example, but the same thing holds true across multiple tables. Every row in a temporal table that is affected in a transaction will have the same start time (and end time) in the corresponding history table. This will allow us to see that before the change was made, policy numbers started with `'AA'`, but now they start with `'IN'`, and the user doesn't have to think that rows 1-10000 have one start time and rows 10001+ have a different start time.

WOW! eBook
www.wowebook.org

Setting/Rewriting History

History rows cannot be modified at all if the table is connected to a table to represent historical rows, but there are two major places where you may need to change history rows:

- *Major mistakes*: As alluded to previously, if you make a mistake and correct it in the base table, it will still be reflected as wrong in the history table. Sometimes, it may be advantageous to fix history so it reflects what was really true. (As I noted in the intro to this main "Temporal/Historical Data" section, temporal tables are probably not where you want to audit changes for security purposes, but rather audit changes for business reasons.)
- *Employing temporal extensions when you upgrade a table to 2016*: Since this is a new feature, and not a new need, many people have already built solutions that keep history using some other techniques (like the trigger I will include in the next section). You can load your own data into the historical table if you need to.

As a very simple example, let's change all of our history to go back to the start of the year 2016. First, let's find the time we started keeping temporal data on the `HumanResources.Employee` table:

```
SELECT MIN(RowStartTime)
FROM   HumanResources.Employee FOR SYSTEM_TIME ALL;
```

This returns (for me, on my 100th+ time of running this script to get it just right).

```
---------------------------
2016-05-10 02:35:49.1
```

This is the time value we will need to create new version rows later. Next, we will turn off versioning, which will turn `HumanResourcesHistory.Employee` into a regular table that can be modified:

```
ALTER TABLE HumanResources.Employee
         SET (SYSTEM_VERSIONING = OFF);
```

The next step is to update all of the rows that have `'2016-05-10 02:35:49.1'` as their start time to `'2016-01-01'`. You will not want to do this for any row's minimum, because that means they were started after versioning was turned on. (And in a real case, you will want to do a lot of research to determine what times make sense for all of the rows, as you will want to figure out what their actual first time of existence was.)

```
--Rows that have been modified
UPDATE HumanResourcesHistory.Employee
SET    RowStartTime = '2016-01-01 00:00:00.0'
WHERE  RowStartTime = '2016-05-10 02:35:49.1'; --value from previous select if you are
                                               --following along in the home game
```

Additionally, you will need to generate history rows for rows that had not been modified yet, as you cannot change the `RowStartTime`:

```
INSERT INTO HumanResourcesHistory.Employee (EmployeeId, EmployeeNumber,
                                            InsurancePolicyNumber, RowStartTime, RowEndTime)
SELECT EmployeeId, EmployeeNumber, InsurancePolicyNumber,
       '2016-01-01 00:00:00.0',
```


WOW! eBook
www.wowebook.org

```
        RowStartTime --use the rowStartTime in the row for the endTime of the history
FROM   HumanResources.Employee
WHERE  NOT EXISTS (SELECT *
                   FROM   HumanResourcesHistory.Employee AS HistEmployee
                   WHERE  HistEmployee.EmployeeId = Employee.EmployeeId);
```

If you have done it correctly, you will have one row per Employee row that you want to go back to the 1st of January returned in the following query:

```
SELECT *
FROM   HumanResourcesHistory.Employee
WHERE  RowStartTime = '2016-01-01 00:00:00.0'
ORDER BY EmployeeId;
```

Which we do:

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
1           A0001          1111111111            2016-01-01 00:00:00.0 2016-05-10 03:08:50.9
2           A0002          2222222222            2016-01-01 00:00:00.0 2016-05-10 03:08:50.9
3           A0003          3333333333            2016-01-01 00:00:00.0 2016-05-10 03:08:50.9
4           A0004          NULL                  2016-01-01 00:00:00.0 2016-05-10 02:46:58.3
5           A0005          NULL                  2016-01-01 00:00:00.0 2016-05-10 02:35:49.1
6           A0006          NULL                  2016-01-01 00:00:00.0 2016-05-10 02:35:49.1
```

Then turn back on system versioning:

```
ALTER TABLE HumanResources.Employee
        SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE = HumanResourcesHistory.Employee));
```

When you run the following:

```
SELECT *
FROM   HumanResources.Employee FOR SYSTEM_TIME AS OF '2016-01-01 00:00:00.0'
ORDER BY EmployeeId;
```

you can see that now your data seems to have existed since the start of 2016, instead of when I was writing this chapter:

```
EmployeeId  EmployeeNumber InsurancePolicyNumber RowStartTime          RowEndTime
----------- -------------- --------------------- --------------------- ---------------------
1           A0001          1111111111            2016-01-01 00:00:00.0 2016-05-10 03:08:50.9
2           A0002          2222222222            2016-01-01 00:00:00.0 2016-05-10 03:08:50.9
3           A0003          3333333333            2016-01-01 00:00:00.0 2016-05-10 03:08:50.9
4           A0004          NULL                  2016-01-01 00:00:00.0 2016-05-10 02:46:58.3
5           A0005          NULL                  2016-01-01 00:00:00.0 2016-05-10 02:35:49.1
6           A0006          NULL                  2016-01-01 00:00:00.0 2016-05-10 02:35:49.1
```


WOW! eBook
www.wowebook.org

You can make other changes to the history while the tables are not paired, but this is definitely one of the easiest. In the downloadable code I have an additional example that will change `EmployeeNumber 'A0005'` to have had insurance since the start of March. This will entail splitting a history in two, so you have one history row for the before image, and another for the after image. It is messy and tedious, so you will want to set up repeatable code processes if you have to modify history in anything other than a very simple manner on a repeating process.

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 orders for products form a hierarchy with customers buying products, which then link to other customers who bought the same products. In the Lego example from earlier in this chapter, if modeled to completion, it 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 briefly 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.

The examples in the text will be generally limited to creating some data and adding data. However, in the downloads you will find a more complete coverage of hierarchies as a companion to the text, with fully fleshed-out code samples to use as a jumping-off point to implement any of the subpatterns presented here. Each method covered here will also include code to

- *Move/reparent a node*: Changing the parent of a node, like changing the manager of an employee
- *Delete a node*: Removing a node from a tree, including any children
- *Aggregate activity for all children of nodes in a tree*: Returning all of the children of every node in the tree, and aggregating all related activities (like sales for a sales territory, as in our main example hierarchy)

The last item will be used as a comparison activity to demonstrate the relative speed of each algorithm. Each method that is represented will include a method to load around a half million nodes into the hierarchy to test performance.

Self-Referencing/Recursive Relationship/Adjacency List

The self-referencing relationship is definitely the easiest method to implement a hierarchy. I covered it a bit back in Chapter 3 when I discussed recursive relationships. They are considered recursive because of the way they are worked with, in both procedural and relational code. In this section, I will cover trees (which are single-parent hierarchies) and then graphs, which allow every node to have multiple parents.

WOW! eBook
www.wowebook.org

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.

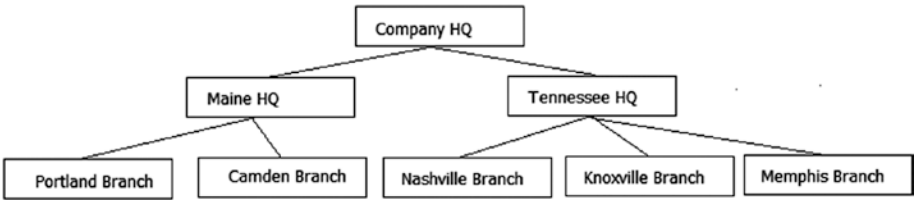


Figure 8-3. *Demonstration company hierarchy*

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. Both use recursion, but the implementation is very different. If you were searching a tree in a functional language, a very common algorithm would be to traverse the tree one node at a time, from the topmost item, down to the lowest in the tree, and then work your way around to all of the nodes. This is generally done using a recursive algorithm, based on the ordering of the items in the tree. In Figure 8-4, I show this for the left side of the tree.

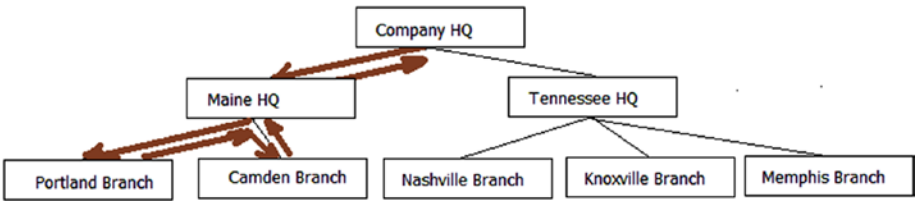


Figure 8-4. *Sample tree structure searched depth first*

This is referred to as a depth-first search and is fast when the language is optimized for single-instance-at-a-time access, particularly when you can load the entire tree structure into RAM. If you attempted to implement this using T-SQL, you would find that it is obnoxiously 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.

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.

WOW! eBook
www.wowebook.org

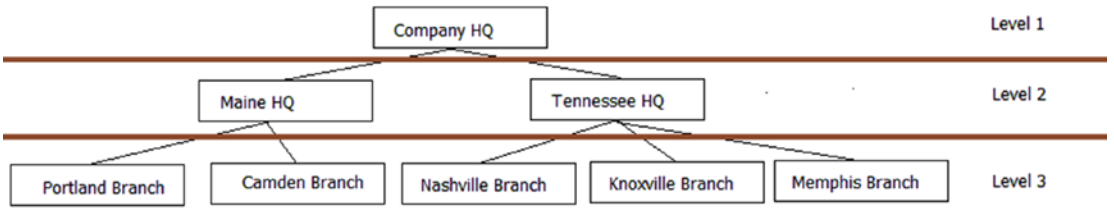


Figure 8-5. *Sample tree structure with levels*

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. In SQL, we will use a recursive CTE (common table expression), where the recursion isn't based on ordering, but rather the anchor SQL referencing the object it is a part of.

To demonstrate working with adjacency list tables, let's create a table to 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.

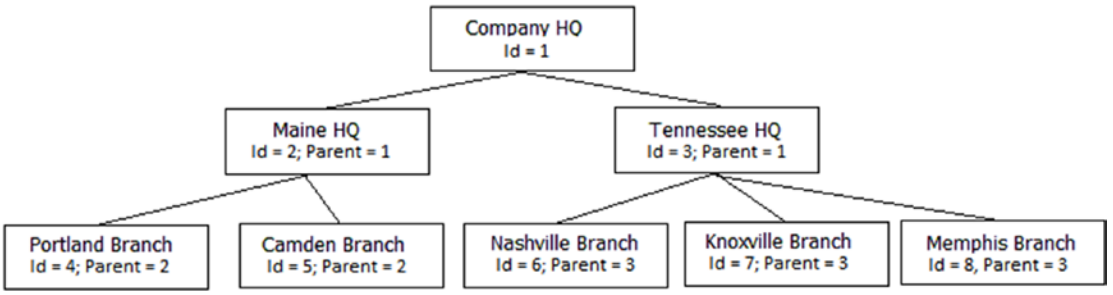


Figure 8-6. *Diagram of basic adjacency list*

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, 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),
```


WOW! eBook
www.wowebook.org

```
       (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 the following:

```
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
7           Portland Branch     2
8           Camden Branch       2
```

You can traverse the data from child to parent fairly easy using the key structures. 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 once you have wrapped your head around it a few times. Recursive CTEs are not always the easiest code to follow:

```
--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,
            CONCAT(hierarchy,'\',Company.CompanyId) AS hierarchy
```


WOW! eBook
www.wowebook.org

```
     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
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 to the data used by what will be called 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:

WOW! eBook
www.wowebook.org

```
        --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 change `@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
```

The `hierarchy` column now shows the relationship of the row to the starting point in the query, not its 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 (100 by default, and controlled per statement via the `MAXRECURSION` query option; set to 0 and no limit is applied) and you stop after `MAXRECURSION` iterations rather than failing, and hence never noticing.

Graphs (Multiparent Hierarchies)

Querying graphs is a very complex topic that is well beyond the scope of this book and chapter. This section provides a brief overview. There are two types of graph-querying problems that you may come up against:

- *Directed*: While the node may have more than one parent, there may not exist cycles in the graph. This is common in a graph like a bill of materials/product breakdown. In this case, you can process a slice of the graph from parent to children exactly like a tree. As no item can be a child and a grandparent of the same node.

WOW! eBook
www.wowebook.org

- *Undirected*: The far more typical graph. An example of an undirected graph is seen in a representation of actors and movies (not to mention directors, staff, clips, etc.). It is often noted in the problem of "Seven Degrees of Kevin Bacon" whereas Kevin Bacon can be linked to almost anyone in seven steps. He was in a movie with Actor1, who was in a movie with Actor2, who was in a movie with Actor3. And Actor3 may have been in two movies with Actor1 already and so forth. When processing a graph, you have to detect cycles and stop processing.

Let's look at a bill of materials. 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 will look at several combinations of object configurations to tweak what your database can model). 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 PKPart PRIMARY KEY,
        PartNumber char(5) NOT NULL CONSTRAINT AKPart UNIQUE,
        Name    varchar(20) NULL
);
```

Then, we load in some simple data:

```
INSERT INTO Parts.Part (PartId, PartNumber,Name)
VALUES (1,'00001','Screw Package'),(2,'00002','Piece of Wood'),
       (3,'00003','Tape Package'),(4,'00004','Screw and Tape'),
       (5,'00005','Wood with Tape') ,(6,'00006','Screw'),(7,'00007','Tape');
```

Next, we create a table to hold the part containership setup:

```
CREATE TABLE Parts.Assembly
(
       PartId   int
            CONSTRAINT FKAssembly$contains$PartsPart
                           REFERENCES Parts.Part(PartId),
       ContainsPartId   int
            CONSTRAINT FKAssembly$isContainedBy$PartsPart
                           REFERENCES Parts.Part(PartId),
            CONSTRAINT PKAssembly PRIMARY KEY (PartId, ContainsPartId)
);
```

First, set up the two packages of screw and tape:

```
INSERT INTO PARTS.Assembly(PartId,ContainsPartId)
VALUES (1,6),(3,7);
```


WOW! eBook
www.wowebook.org

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);
```

Finally, you can do the same thing for the Wood with Tape part:

```
INSERT INTO Parts.Assembly(PartId,ContainsPartId)
VALUES (5,2),(5,3);
```

Now you can take any part in the hierarchy, and use the same recursive CTE-style algorithm and pull out the tree of data you are interested in. In the first go I will get the 'Screw and Tape' assembly which is PartId=4:

```
--getting the children of a row (or ancestors with slight mod to query)
DECLARE @PartId int = 4;

;WITH partsHierarchy(PartId, ContainsPartId, treelevel, hierarchy,nameHierarchy)
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 NULL  AS PartId, PartId AS ContainsPartId,
            1 AS treelevel,
            CAST(PartId AS varchar(max)) as hierarchy,
            --added more textual hierarchy for this example
            CAST(Name AS varchar(max)) AS nameHierarchy
     FROM   Parts.Part
     WHERE PartId=@PartId

     UNION ALL

     --joins back to the CTE to recursively retrieve the rows
     --note that treelevel is incremented on each iteration
     SELECT Assembly.PartId, Assembly.ContainsPartId,
            treelevel + 1 as treelevel,
            CONCAT(hierarchy,'\',Assembly.ContainsPartId) AS hierarchy,
            CONCAT(nameHierarchy,'\',Part.Name) AS nameHierarchy
     FROM   Parts.Assembly
                          INNER JOIN Parts.Part
                                ON Assembly.ContainsPartId = Part.PartId
               INNER JOIN partsHierarchy
                 ON Assembly.PartId= partsHierarchy.ContainsPartId
)
SELECT PartId, nameHierarchy, hierarchy
FROM partsHierarchy;
```


WOW! eBook
www.wowebook.org

This returns

```
PartId      nameHierarchy                       hierarchy
----------- ----------------------------------- ------------
NULL        Screw and Tape                      4
4           Screw and Tape\Screw Package        4\1
4           Screw and Tape\Tape Package         4\3
3           Screw and Tape\Tape Package\Tape    4\3\7
1           Screw and Tape\Screw Package\Screw  4\1\6
```

Change the variable to 5, and you will see the other part we configured:

```
PartId      nameHierarchy                       hierarchy
----------- ----------------------------------- ------------
NULL        Wood with Tape                      5
5           Wood with Tape\Piece of Wood        5\2
5           Wood with Tape\Tape Package         5\3
3           Wood with Tape\Tape Package\Tape    5\3\7
```

As you can see, the tape package is repeated from the other part configuration. I won't cover dealing with cycles in graphs in the text (it will be a part of the extended examples), but the biggest issue when dealing with cyclical graphs is making sure that you don't double count data because of the cardinality of the relationship.

Implementing the Hierarchy Using the hierarchyId Type

In addition to the fairly standard adjacency list implementation, there is also a datatype called `hierarchyId` that is a proprietary CLR-based datatype that can be 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, which will not be possible for the `hierarchyId` solution, unless you have already calculated the path). 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. Note the addition of a calculated column that indicates the level in the hierarchy, which will be used by the internals to support breadth-first processing. The surrogate key `CompanyId` is not clustered to allow for a future index. If you do a large amount of fetches by the primary key, you may want to implement the hierarchy as a separate table.

```
CREATE TABLE Corporate.CompanyAlternate
(
    CompanyOrgNode hierarchyId not null
                  CONSTRAINT AKCompanyAlternate UNIQUE,
    CompanyId   int CONSTRAINT PKCompanyAlternate PRIMARY KEY NONCLUSTERED,
    Name        varchar(20) CONSTRAINT AKCompanyAlternate_name UNIQUE,
    OrganizationLevel AS CompanyOrgNode.GetLevel() PERSISTED
);
```


WOW! eBook
www.wowebook.org

You will also want to add an index that includes the level and `hierarchyId` node. Without the calculated column and index (which are not at all intuitively obvious), the performance of this method will degrade rapidly as your hierarchy grows:

```
CREATE CLUSTERED INDEX Org_Breadth_First
         ON Corporate.CompanyAlternate(OrganizationLevel,CompanyOrgNode);
```

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.CompanyAlternate (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 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), with comments to explain how the code works:

```
CREATE PROCEDURE Corporate. CompanyAlternate$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. CompanyAlternate
                               WHERE  CompanyID = @ParentCompanyId)
         IF  @parentCompanyOrgNode IS NULL
           BEGIN
                THROW 50000, 'Invalid parentCompanyId passed in',1;
                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.CompanyAlternate (UPDLOCK) --compatible 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
```


WOW! eBook
www.wowebook.org

```
      --the one passed in.  Since the value was the max in the table, the
      --getDescendant Method returns the next one
      INSERT Corporate.CompanyAlternate  (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.CompanyAlternate$insert @CompanyId = 1, @parentCompanyId = NULL,
--                                 @Name = 'Company HQ'; --already created
exec Corporate.CompanyAlternate$insert @CompanyId = 2, @ParentCompanyId = 1,
                                 @Name = 'Maine HQ';
exec Corporate.CompanyAlternate$insert @CompanyId = 3, @ParentCompanyId = 1,
                                 @Name = 'Tennessee HQ';
exec Corporate.CompanyAlternate$insert @CompanyId = 4, @ParentCompanyId = 3,
                                 @Name = 'Knoxville Branch';
exec Corporate.CompanyAlternate$insert @CompanyId = 5, @ParentCompanyId = 3,
                                 @Name = 'Memphis Branch';
exec Corporate.CompanyAlternate$insert @CompanyId = 6, @ParentCompanyId = 2,
                                 @Name = 'Portland Branch';
exec Corporate.CompanyAlternate$insert @CompanyId = 7, @ParentCompanyId = 2,
                                 @Name = 'Camden Branch';
```

You can see the data in its raw format here:

```
SELECT CompanyOrgNode, CompanyId, Name
FROM   Corporate.CompanyAlternate
ORDER  BY CompanyId;
```

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:

WOW! eBook
www.wowebook.org

```
SELECT CompanyId, OrganizationLevel,
       Name, CompanyOrgNode.ToString() as Hierarchy
FROM   Corporate.CompanyAlternate
ORDER  BY Hierarchy;
```

This can be really useful in queries:

```
companyId   OrganizationLevel  name                hierarchy
----------- ------------------ ------------------- -------------
1           0                  Company HQ          /
2           1                  Maine HQ            /1/
6           2                  Portland Branch     /1/1/
7           2                  Camden Branch       /1/2/
3           1                  Tennessee HQ        /2/
4           2                  Knoxville Branch    /2/1/
5           2                  Memphis Branch      /2/2/
```

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:

```
DECLARE @CompanyId int = 3;
SELECT Target.CompanyId, Target.Name, Target.CompanyOrgNode.ToString() AS Hierarchy
FROM   Corporate.CompanyAlternate AS Target
               JOIN Corporate.CompanyAlternate 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.CompanyAlternate AS Target
               JOIN Corporate.CompanyAlternate AS SearchFor
                       ON SearchFor.CompanyId = @CompanyId
                          and SearchFor.CompanyOrgNode.IsDescendantOf
                                                  (Target.CompanyOrgNode) = 1;
```


WOW! eBook
www.wowebook.org

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` has several 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:

- *Path technique*: In this method, which is similar internally to using `hierarchyId`, (though without the methods on a datatype to work with,) 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.

Path Technique

The path technique is pretty much the manual version of the `hierarchyId` method. In it, you store the path from the parent to the child. Using our hierarchy that we have used so far, 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, so it will be essential that the value is immutable. In Figure 8-7, I have included a diagram of the hierarchy implemented with the path value set for our design.

WOW! eBook
www.wowebook.org

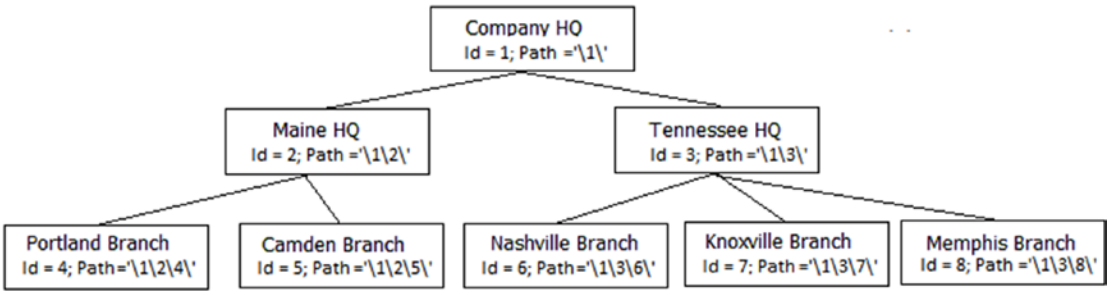


Figure 8-7. *Sample hierarchy diagram with values for the path technique*

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\'`.

One of the great things about the path method is that it readily uses indexes, because most of the queries you will make will use the left side of a string. So up to SQL Server 2014, as long as your path could stay below 900 bytes, your performance is generally awesome. In 2016, the max keylength has been increased to 1,700 bytes, which is great, but if your paths are this long, you will only end up with four keys per page, which is not going to give amazing performance (indexes will be covered in detail in Chapter 10).

Nested Sets

One of the more clever methods of dealing with hierarchies 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. It is also a favorite method of Joe Celko, who has written a book about hierarchies named *Joe Celko's Trees and Hierarchies in SQL for Smarties* (Morgan Kaufmann, 2004); check it out (now in its second edition, 2012) 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.

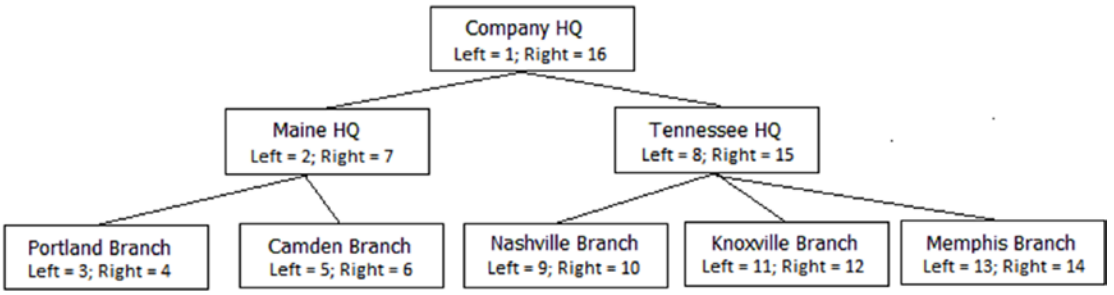


Figure 8-8. *Sample hierarchy diagram with values for the nested sets technique*

WOW! eBook
www.wowebook.org

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 integer comparisons. To find the parents of `Maine HQ`, you simply need to look for the case `WHERE Left < 2 and Right > 7`.

Adding a node has a rather 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 fairly stable. Going back to our adjacency list implementation, shown in Figure 8-9, assume we already have this implemented in SQL.

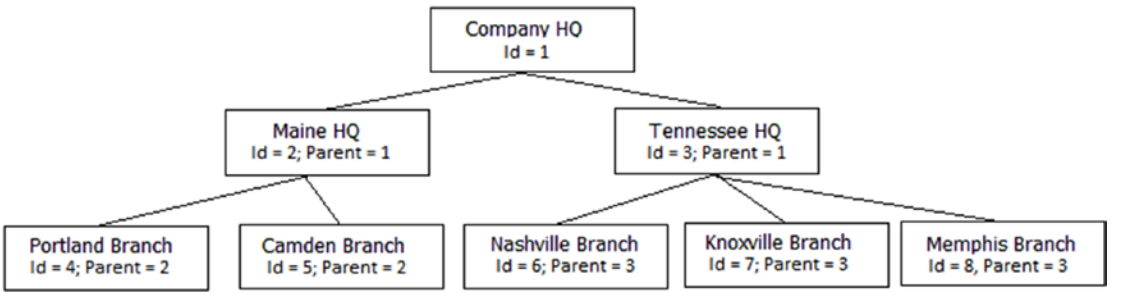


Figure 8-9. *Sample hierarchy diagram with values for the adjacency list technique repeated for the Kimball helper table method*

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`.

WOW! eBook
www.wowebook.org

The code to implement this structure is basically a slightly modified version of the recursive CTE used in our first hierarchy example. It may take a few minutes to rebuild for a million nodes.

The obvious downfall of this method is simple. It is somewhat costly to maintain if the structure is modified frequently. 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 as 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 on a way to store binary data in SQL Server, there are two broadly characterizable ways that are available:

- Storing a path reference to the file data
- Storing the binaries using SQL Server's storage engine

In early editions of this book, 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 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 2012, the picture improved 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.

Over the course of time, the picture hasn't changed terribly. I generally separate the choice between the two possible ways to store binaries into one primary simple reason to choose one or the other: transactional integrity. If you require transaction integrity, you use SQL Server's storage engine, regardless of the cost you may incur. If transaction integrity isn't tremendously important, you probably will want to use the file system. For example, if you are 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 is a consideration, but if you need 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.

If you are going to store large objects in SQL Server, you will usually want to use filestream, particularly if your files are of fairly large size. It is suggested that you definitely consider filestream if your binary objects will be greater than 1MB, but recommendations change over time. Setting up filestream access is pretty easy; first, you enable filestream access for the server. For details on this process, check the Books Online topic "Enable and Configure FILESTREAM" (`https://msdn.microsoft.com/en-us/library/cc645923.aspx`). The basics to enable filestream (if you did not already do this during the process of installation) are to 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.

WOW! eBook
www.wowebook.org

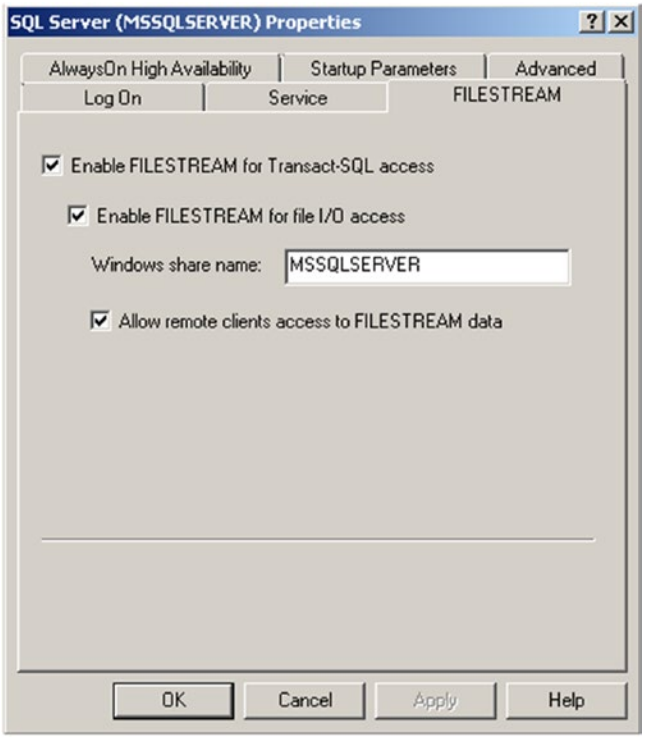


Figure 8-10. *Configuring the server for filestream access*

The Windows share name will be used to access filestream data via the API, as well as using a filetable later in this chapter. Later in this section, there will be additional configurations based on how the filestream data will be accessed. Start by enabling filestream access for the server using `sp_configure filestream_access_level` of either `1` (T-SQL access) or `2` (T-SQL and Win32 access). We will be using both methods, so I will use the latter:

```
EXEC sp_configure filestream_access_level 2;
RECONFIGURE;
```

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 database
GO
USE FileStorageDemo;
GO
--will cover filegroups more in the chapter 10 on structures
ALTER DATABASE FileStorageDemo ADD
        FILEGROUP FilestreamData CONTAINS FILESTREAM;
```


WOW! eBook
www.wowebook.org

■ **Tip** There are caveats with using filestream data in a database that also needs to use snapshot isolation level or that implements the `READ_COMMITTED_SNAPSHOT` database option. Go to `SET TRANSACTION ISOLATION LEVEL` statement (covered in Chapter 11) documentation here https://msdn.microsoft.com/en-us/library/ms173763.aspx for more information.

Next, add a "file" to the database that is actually a directory for the filestream files (note that the directory should not exist before executing the following statement, but the directory, in this case, `c:\sql`, must exist or you will receive an error):

```
ALTER DATABASE FileStorageDemo ADD FILE (
       NAME = FilestreamDataFile1,
       FILENAME = 'c:\sql\filestream') --directory cannot yet exist and SQL account must have
                                       --access to drive.
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 SCHEMA Demo;
GO
CREATE TABLE Demo.TestSimpleFileStream
(
        TestSimpleFilestreamId INT NOT NULL
                       CONSTRAINT PKTestSimpleFileStream PRIMARY KEY,
        FileStreamColumn VARBINARY(MAX) FILESTREAM NULL,
        RowGuid uniqueidentifier NOT NULL ROWGUIDCOL DEFAULT (NEWID())
                       CONSTRAINT AKTestSimpleFileStream_RowGuid UNIQUE
)       FILESTREAM_ON FilestreamData;
```

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 a simple query:

```
INSERT INTO Demo.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   Demo.TestSimpleFilestream;
```

I won't go any deeper into filestream manipulation here, 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.

WOW! eBook
www.wowebook.org

In SQL Server 2012, we got 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 that is accessible from Windows Explorer.

Enable and set up filetable style filestream in the database as follows:

```
ALTER DATABASE FileStorageDemo
        SET FILESTREAM (NON_TRANSACTED_ACCESS = FULL,
                        DIRECTORY_NAME = N'ProSQLServerDBDesign');
```

The setting NON_TRANSACTED_ACCESS lets you set whether or not users can change data when accessing the data as a Windows share, such as opening a document in Word. 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 Demo.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 a directory for Project 1:

```
INSERT INTO Demo.FiletableTest(name, is_directory)
VALUES ( 'Project 1', 1);
```

Then, you can view this data in the table:

```
SELECT stream_id, file_stream, name
FROM   Demo.FileTableTest
WHERE  name = 'Project 1';
```

This will return (though with a different stream_id)

```
stream_id                            file_stream                          name
------------ ------------9BCB8987-1DB4-E011-87C8-000C29992276          Project 1
```


WOW! eBook
www.wowebook.org

stream_id is 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 Demo.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. (The directory hierarchy is built on hierarchyId. In the downloads for hierarchies from the earlier section, you can see more details of the methods you can use here as well as your own hierarchies.)

```
UPDATE Demo.FiletableTest
SET    path_locator = path_locator.GetReparentedValue( path_locator.GetAncestor(1),
       (SELECT path_locator FROM Demo.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 my case, the name of my VM is WIN-8F59BO5AP7D, so the share is \\WIN-8F59BO5AP7D\MSSQLSERVER\ProSQLServerDBDesign, which is my computer's name, the MSSQLSERVER we set up in Configuration Manager, and ProSQLServerDBDesign 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    Demo.FileTableTest
WHERE   name = 'Project 1'
  AND   parent_path_locator is NULL
  AND   is_directory = 1;
```

This returns the following:

```
FilePath
-------------------------------------------------------------------------------
\\WIN-8F59BO5AP7D\MSSQLSERVER\ProSQLServerDBDesign\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, and you may need to set up your firewall to allow access as well.

WOW! eBook
www.wowebook.org

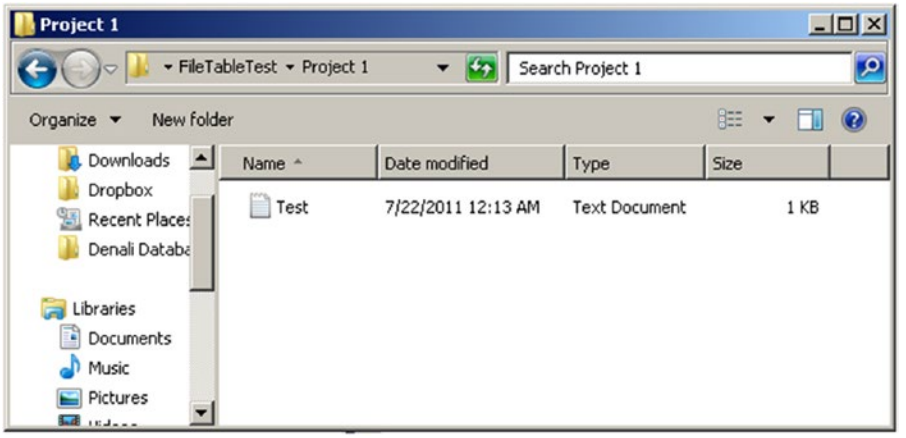


Figure 8-11. *Filetable directory opened in Windows Explorer*

From here, I would suggest you drop a few files in the directory from your local drive 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.

■ **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. You don't have any row-level security over access, so if you need more security, you may need a table per security needs, which may not be optimum for more than a few use cases.

So, mechanics aside, consider the four various 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 (RBS) 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. For more information, see: `https://msdn.microsoft.com/en-us/library/gg638709.aspx`.

WOW! eBook
www.wowebook.org

Each of these methods 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.

- *Transactional integrity*: Guaranteeing that the image is stored and remains stored is far easier if it is managed by the storage engine, either as a filestream or as a binary value, than 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 if you need the file modification to occur along with other changes as a transaction, you will need to use the filestream API.
- *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 (as it needs control over the directories to provide integrity). For filestream column use, the data, just like a normal filegroup, must be transactionally safe for utilization.
- *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 photo 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.

For three quick examples, consider a movie rental database (online these days, naturally). 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

WOW! eBook
www.wowebook.org

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.

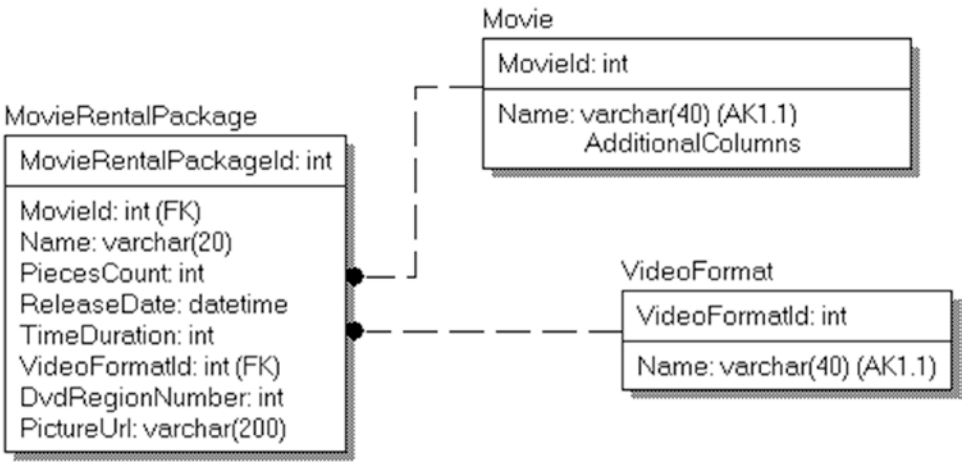


Figure 8-12. *MovieRentalPackage table with PictureUrl datatype set as a path to a file*

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 to get the picture that you would display for the movie's catalog entry:

```
SELECT '<img src = "' + MovieRentalPackage.PictureUrl + '">', ...
FROM    Movies.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 digital photos of customers and comparing the photos to the customers whenever they 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.

WOW! eBook
www.wowebook.org

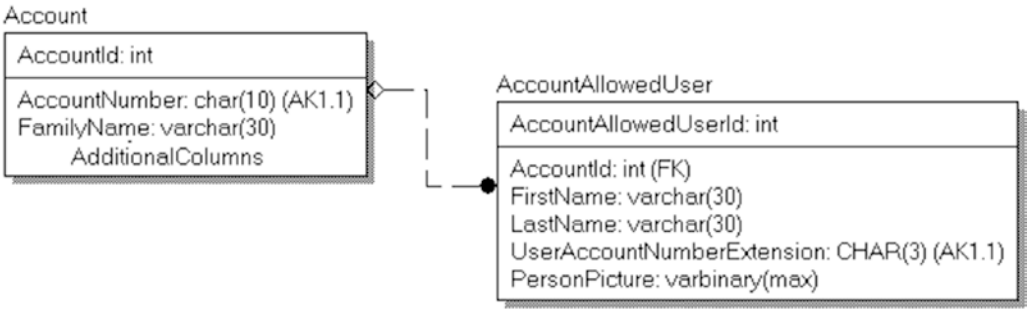


Figure 8-13. *Customer table with picture stored as data in the table*

At this point, assume 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 is whether to employ filestreams. The big question with regard to this decision is whether your API supports filestreams. If so, this would likely be a very good place to make use of them. Size could play a part in the choice too, though security pictures could 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 key; even though it is technically a unique constraint in implementation, you can reference an alternate key).

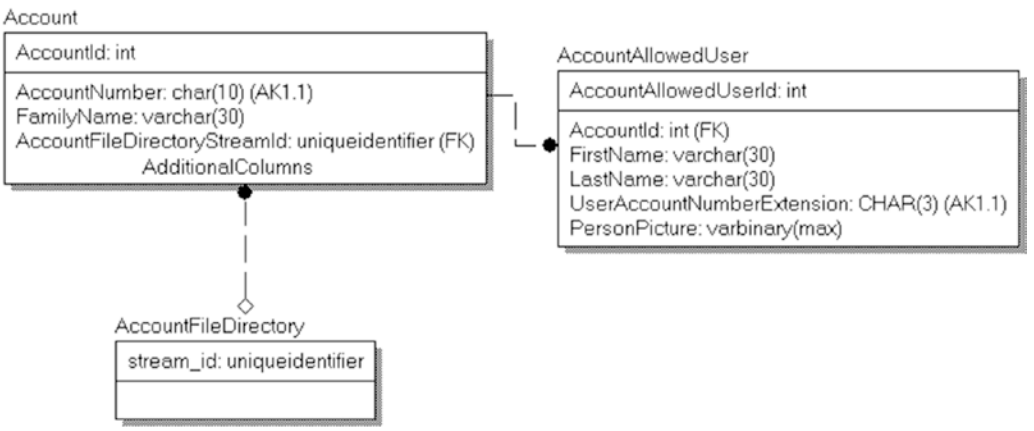


Figure 8-14. *Account model extended with an AccountFileDirectory*

WOW! eBook
www.wowebook.org

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 not only will be very usable for the programmer and user alike but will 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.

Generalization

Designing is often discussed as an art form, and that is what this topic is about. When designing a set of tables to represent some real-world activity, how specific should your tables be? 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, another for the swimming class, and 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 look exactly the same, you would start to notice that every table is used for basically the same thing: assign an instructor, sign up kids to attend, add a description, and so forth. Rather than the system being about each activity, requiring you to model each of the different activities as being different from one another, what you would truly need to do is model the *abstraction* of a camp activity. On the other hand, while the primary focus of the design would be the management of the activity, you might discover that some extended information is needed about some or all of the classes. Generalization is about making objects as general as possible, employing a pattern like subclassing to tune in the best possible solution.

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. In our 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 (or even worse, one general-purpose bucket of a table with a `varchar(max)` column where all of the desired information is shoveled into).

In those cases, you can consider using a subclassed entity for certain entities. Take the camp activity model. You could include the generalized table for the generic `CampActivity`, in which you would associate students 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 (in unshown related tables), as shown in Figure 8-15.

WOW! eBook
www.wowebook.org

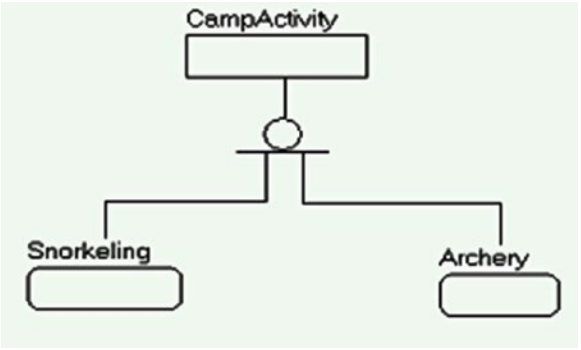


Figure 8-15. *Extending generalized entity with specific details as required*

As a coded example, we will look at a home inventory system. Suppose we have a client who wants to create an inventory of the various types of stuff in their house, or at least everything valuable, for insurance purposes. So, should we simply design a table for each type of item? That seems like too much trouble, because for almost everything the client will simply have a description, picture, value, and a receipt. On the other hand, a single table, generalizing all of the items in the client's house down to a single list, seems like it might not be enough for items that they 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 the client has a basic list of the home inventory, but can also print a list of jewelry alone with extra detail or print a list of electronics with their identifying information.

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 PKItem PRIMARY KEY,
        Name    varchar(30) NOT NULL CONSTRAINT AKItemName 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
);
```

As you can see, I included two columns for holding an image of the receipt and a picture of the item. As 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 varbinary 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),
```


WOW! eBook
www.wowebook.org

```
       ('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 organized how we really need it to be, because in realistic usage, we will probably need some of the specific data from the descriptions to be easier to access:

```
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...
```

At this point, we look at our data and reconsider the design. The two pieces of furniture are fine as listed. We have a picture and a brief description. For the other three 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 the client 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 PKJeweleryItem PRIMARY KEY
                    CONSTRAINT FKJeweleryItem$Extends$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 PKElectronicItem PRIMARY KEY
                    CONSTRAINT FKElectronicItem$Extends$InventoryItem
                                        REFERENCES Inventory.Item(ItemId),
```


WOW! eBook
www.wowebook.org

```
        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 the electronics for which the client doesn't have a serial number listed and needs to provide 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:

```
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, we see the more generic list with names that are more specifically for the person maintaining the list:

```
SELECT Name, Type, Description
FROM   Inventory.Item;
```


WOW! eBook
www.wowebook.org

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, we 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 to want to see a complete inventory with the specific information, since this is truly the natural way to think of the data and is 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 CONCAT('Brand:', COALESCE(BrandName,'_______'),
                 ' Model:',COALESCE(ModelNumber,'________'),
                 ' SerialNumber:', COALESCE(SerialNumber,'_______'))
          WHEN 'Jewelery'
            THEN CONCAT('QualityLevel:', QualityLevel,
                 ' Appraiser:', COALESCE(AppraiserName,'_______'),
                 ' AppraisalValue:', COALESCE(Cast(AppraisalValue as
                   varchar(20)),'_______'),
                 ' AppraisalYear:', COALESCE(AppraisalYear,'____'))
            ELSE '' END as ExtendedDescription
FROM   Inventory.Item --simple outer joins because every not item will have extensions
                      --but they will only have one if any extension
           LEFT OUTER JOIN Inventory.ElectronicItem
                ON Item.ItemId = ElectronicItem.ItemId
           LEFT OUTER JOIN Inventory.JeweleryItem
                ON Item.ItemId = JeweleryItem.ItemId;
```


WOW! eBook
www.wowebook.org

This returns a formatted description, and visually shows missing information:

```
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 would normally get the blame. But the truth is, if you really listen to the user's needs and model them correctly, you will naturally generalize your objects. However, it is still a good thing to look 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, having a column for that data almost certainly 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

Try as one might, it is nearly impossible to get a database design done perfectly, especially for unseen future needs. Users need to be able to morph their schema slightly at times to add some bit of information that they didn't realize would exist, and doesn't meet the needs of changing the schema and user interfaces. So we need to find some way to provide a method of tweaking the schema without changing the interface. The biggest issue is the integrity of the data that users want to store in this database, in that it is very rare that they'll want to store data and not use it to make decisions. In this section, I will explore a couple of the common methods for enabling the end user to expand the data catalog.

As I have tried to make clear throughout the book so far, relational tables are not meant to be flexible. T-SQL as a language is not made for flexibility (at least not from the standpoint of producing reliable databases that produce expected results and producing acceptable performance while protecting the quality of the data, which, as 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 users the flexibility they demand, along with methods to deal with this data in a manner that feels good to them.

WOW! eBook
www.wowebook.org

■ **Note** I will specifically speak only of methods that allow you to work with the relational engine in a seminatural manner. One method I won't cover is using a normal XML column. The second method I will show actually uses an XML-formatted basis for the solution in a far more natural solution.

The methods I will demonstrate are as follows:

- 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 PKEquipment PRIMARY KEY,
    EquipmentTag varchar(10) NOT NULL
          CONSTRAINT AKEquipment 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. One anti-pattern I won't demonstrate is what I call the "Big Ol' Set of Generic Columns." Basically, it involves adding multiple columns to the table as part of the design, as in the following variant of the `Equipment` table:

```
CREATE TABLE Hardware.Equipment
(
    EquipmentId int NOT NULL
          CONSTRAINT PKHardwareEquipment PRIMARY KEY,
    EquipmentTag varchar(10) NOT NULL
          CONSTRAINT AKHardwareEquipment UNIQUE,
    EquipmentType varchar(10),
    UserDefined1 sql_variant NULL,
    UserDefined2 sql_variant NULL,
    ...
    UserDefinedN sql_variant NULL
);
```


WOW! eBook
www.wowebook.org

I definitely don't favor such a solution because it hides what kind of values are in the added columns, and is often abused because the UI is built to have generic labels as well. Such implementations rarely turn out well for the person who needs to use these values at a later point in time.

Entity-Attribute-Value (EAV)

The first recommended method of implementing user-specified data is the entity-attribute-value (EAV) method. These are also known by a few different names, such as property tables, loose schemas, or open schema. This technique is often considered the default method of implementing a table to allow users to configure their own storage.

The basic idea is to have another related attribute table associated with the table you want to add information about. 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 8-16.

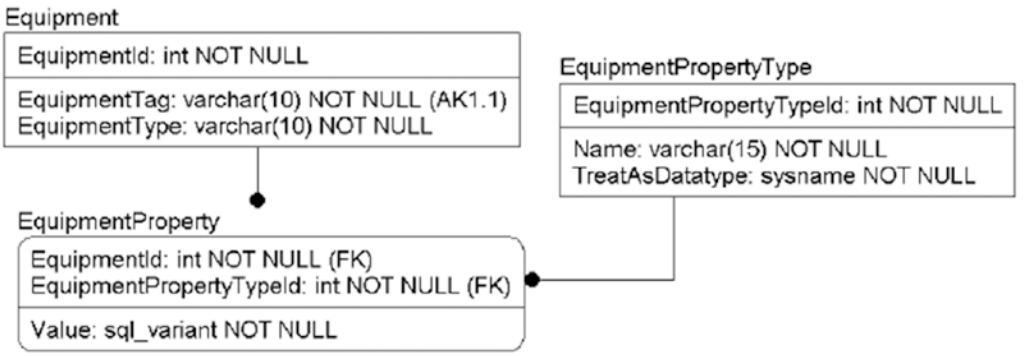


Figure 8-16. *Property schema for storing equipment properties with unknown attributes*

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 on by the users, but still have a modicum of data integrity. 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.

What makes this method desirable to programmers is that you can create a user interface that is just a simple list of attributes to edit. Adding a new property is simply another row in a database. Even the solution I will provide here, with some additional data control, is really easy to provide a UI for.

To create this solution, I will create an `EquipmentPropertyType` table and add a few types of properties:

```
CREATE TABLE Hardware.EquipmentPropertyType
(
    EquipmentPropertyTypeId int NOT NULL
        CONSTRAINT PKEquipmentPropertyType PRIMARY KEY,
    Name varchar(15)
        CONSTRAINT AKEquipmentPropertyType UNIQUE,
    TreatAsDatatype sysname NOT NULL
);
```


WOW! eBook
www.wowebook.org

```
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, which 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 either to use a character string–type value (requiring the caller/user to convert to a string representation of all values) or to have multiple columns, one for each possible/supported datatype. Both of these options and using sql_variant all have slight difficulties, but I tend to use sql_variant for truly unknown types of data because data is stored in its native format and is usable in some ways in its current format (though in most cases you will need to cast the data to some datatype to use it). In the definition of the property, I will also include the datatype that I expect the data to be, and in my insert procedure, I will test the data to make sure it meets the requirements for a specific datatype.

```
CREATE TABLE Hardware.EquipmentProperty
(
    EquipmentId int NOT NULL
      CONSTRAINT FKEquipment$hasExtendedPropertiesIn$HardwareEquipmentProperty
           REFERENCES Hardware.Equipment(EquipmentId),
    EquipmentPropertyTypeId int
      CONSTRAINT FKEquipmentPropertyTypeId$definesTypesFor$HardwareEquipmentProperty
           REFERENCES Hardware.EquipmentPropertyType(EquipmentPropertyTypeId),
    Value sql_variant,
    CONSTRAINT PKEquipmentProperty 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 it to the datatype the user configured for the property. (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 all examples in the book, to keep the samples cleaner, 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;
```


WOW! eBook
www.wowebook.org

```
        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) =
           CONCAT(' DECLARE @value sql_variant
                    SELECT  @value = CAST(VALUE AS ', @TreatASDatatype, ')
                    FROM    Hardware.EquipmentProperty
                    WHERE   EquipmentId = ', @EquipmentId, '
                      and   EquipmentPropertyTypeId = ' ,
                           @EquipmentPropertyTypeId);

        EXECUTE (@validationQuery);
      COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
         IF @@TRANCOUNT > 0
             ROLLBACK TRANSACTION;

         DECLARE @ERRORmessage nvarchar(4000)
         SET @ERRORmessage = CONCAT('Error occurred in procedure ''',
                  OBJECT_NAME(@@procid), ''', Original Message: ''',
                  ERROR_MESSAGE(),''' Property:''',@EquipmentPropertyName,
                 ''' Value:''',cast(@Value as nvarchar(1000)),'''');
      THROW 50000,@ERRORMessage,16;
      RETURN -100;

     END CATCH;
```

So, if you try to put in an invalid piece of data such as

```
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$Insert, Line 49
Error occurred in procedure 'EquipmentProperty$Insert', Original Message: 'Error converting
data type varchar to numeric.'. Property:'Width' Value:'Claw'
```


WOW! eBook
www.wowebook.org

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
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
```


WOW! eBook
www.wowebook.org

```
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             8.4       2
HANDSAW      Saw           NULL             7         1
POWERDRILL   PowerTool     NULL             12.1      6
```

If you execute this on your own in the "Results to Text" mode in SSMS, 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 each column ahead of execution time. In the following extension, I have used `XML PATH` to output the different properties to different columns, starting with `MAX`. (This is a common SQL Server 2005 and later technique for converting rows to columns. Do a web search for "convert rows to 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);
```


WOW! eBook
www.wowebook.org

Executing this will get you the following (which is exactly what was returned in the last results, but you will notice a major difference if you execute this code yourself):

```
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 specified in the `EquipmentPropertyType` table, not as 8,000-character values (that is a lot of little minus signs under each heading to delete). You could expand this code further if you wanted to limit the domain even further than just by datatype, but it definitely will complicate matters.

■ **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 store data.

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 is 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.

■ **Tip** If you build products to ship to customers, 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.

To implement this method, for the most part we will make use of *sparse columns*, 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 storage 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

WOW! eBook
www.wowebook.org

allow you to add many columns to the table (the maximum is 30,000!), 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. Because of the concept of a column set, this solution will allow you to build a UI that doesn't know all of the structure along with a typical SQL solution.

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:

- The spatial types
- `rowversion/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 OWNER
AS
  --note: I did not include full error handling for clarity
  DECLARE @query nvarchar(max);

 --check for column existence
 IF NOT EXISTS (SELECT *
                FROM   sys.columns
                WHERE  name = @propertyName
```


WOW! eBook
www.wowebook.org

```
                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',1;
```

Now, any user to whom 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;
```

This 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
```


WOW! eBook
www.wowebook.org

One thing that is so much more powerful about 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:

```
ALTER TABLE Hardware.Equipment
 ADD CONSTRAINT CHKEquipment$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 "CHKEquipment$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;
```


WOW! eBook
www.wowebook.org

This 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
```

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, or if you are trying to build a UI that morphs to the data. You can define a *column set*, which is the XML representation of the set of columns 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):

```
ALTER TABLE Hardware.Equipment
    DROP CONSTRAINT CHKEquipment$HammerHeadStyle;
ALTER TABLE Hardware.Equipment
    DROP COLUMN HammerHeadStyle, Length, Width;
```


WOW! eBook
www.wowebook.org

Now, I add a column set, which I will name SparseColumns:

```
ALTER TABLE Hardware.Equipment
  ADD SparseColumns XML COLUMN_SET FOR ALL_SPARSE_COLUMNS;
```

Next, I add back the sparse columns and constraints using my existing procedure:

```
EXEC Hardware.Equipment$addProperty 'Length','numeric(10,2)',1;
EXEC Hardware.Equipment$addProperty 'Width','numeric(10,2)',1;
EXEC Hardware.Equipment$addProperty 'HammerHeadStyle','varchar(30)',1;
GO
ALTER TABLE Hardware.Equipment
 ADD CONSTRAINT CHKEquipment$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 (or also insert) 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';
```


WOW! eBook
www.wowebook.org

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
```

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`. So if you wanted to index the `HammerHeadStyle` for the hammer type rows, you might add the following index (preceded by the settings that must be turned on before creating an index on the XML-based column set):

```
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT, QUOTED_IDENTIFIER,
ANSI_NULLS ON
GO
CREATE INDEX HammerHeadStyle_For_ClawHammer ON Hardware.Equipment (HammerHeadStyle) WHERE
EquipmentType = 'Hammer'
```

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, this method is 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

For every good practice put forth to build awesome structures, there come many that fail to work. In this section, I will outline four of these practices that are often employed by designers and implementers, some novice and some experienced, 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.

WOW! eBook
www.wowebook.org

- *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 leads 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:

- *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. Fifth is rare to violate, but if you do you will know why it is interesting as well. (Chapter 5 covers normalization in depth.)
- *Poor domain choices*: Lots of database designers 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 and 6.)
- *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 nine ASCII characters long, it is best to use a `char(9)` column to store that data. Too often a database might 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 is undecipherable values. Code such as `WHERE status = 1` will pepper the code you no doubt discover using SQL Server Profiler, and you as the data developer end up scratching your head in wonderment as to what 1 represents (and then what 2, 3, 5, and 282 represent. Oh yeah, and what happened to 4?). Of course, the reason for this is that the developers don't think of the database as a primary data resource to not only store data, but also will be queried by people other than the code they have written. It is simply thought of as 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 = 2, CONST_BarelyActive = 3, CONST_Asleep = 5,
CONST_Dead = 282);
```


WOW! eBook
www.wowebook.org

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 that mirrors the `CONST` structure. Have a table with all possible values.

For the latter, your table could use the descriptive values as the domain, or you could 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:

- Only have values that can be deciphered using the database:
  - Foreign key to a domain 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 (or at least looks the same shape) in many cases, 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 catch-all table should simplify the design, right? That sounds logical, but at one time giving Adam Sandler 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 indicated in Figure 8-17?

WOW! eBook
www.wowebook.org

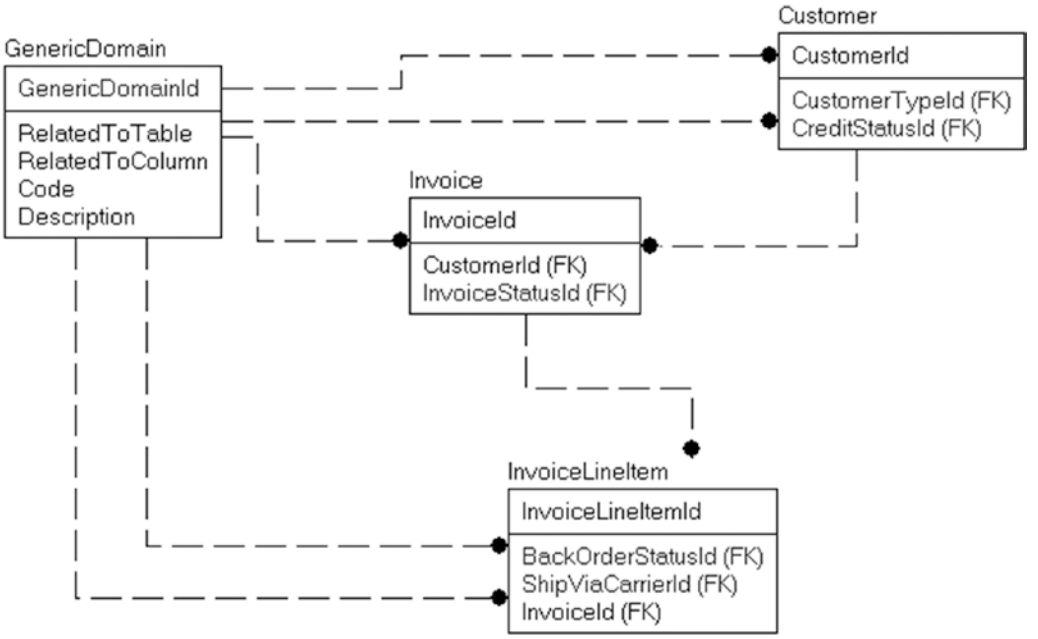


Figure 8-17. *One multiuse domain table*

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';

--NOTE: This code is not part of the downloads, nor are the tables for the examples in
--this anti-pattern section.
```

It comes down to the problem of mixing apples with oranges. When you want to make apple pie, you have to separate 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. 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

WOW! eBook
www.wowebook.org

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, as we explored when defining domains, it is the same domain used in multiple places, 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 8-18.

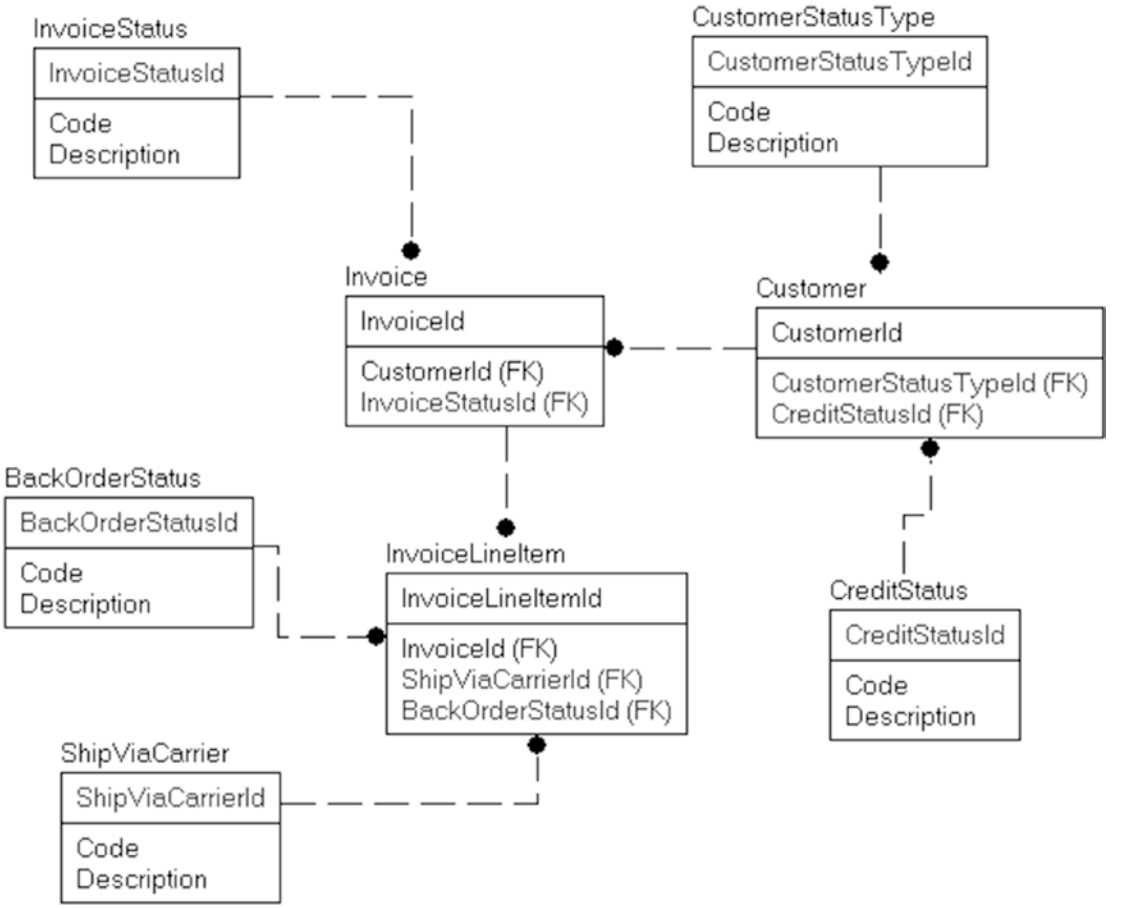


Figure 8-18. *One domain table per purpose*

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. What makes it even more time to implement is that you will actually be able to implement the foreign key constraints to protect the values in the tables no matter what the values are for the Id columns. The fact is, there are quite a few tremendous gains to be had:

- *Using the data in a query is much easier and self-documenting*:

```
SELECT *
FROM Customer
  JOIN CustomerType
```


WOW! eBook
www.wowebook.org

```
  ON Customer.CustomerTypeId = CustomerType.CustomerTypeId
JOIN CreditStatus
  ON  Customer.CreditStatusId = CreditStatus.CreditStatusId
```

- *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. In a very large table, it could get to the point where a scan of a larger domain table could get costly when only a very small number of rows is needed.
- *You can still make the data look like one table for the domain management 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.

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.

Generic Key References

In an ideal situation, one table is related to another via a key. However, it is entirely possible to have a table that has a foreign 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`. 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 over-communicate with a customer!). You might logically draw it up like in Figure 8-19.

WOW! eBook
www.wowebook.org

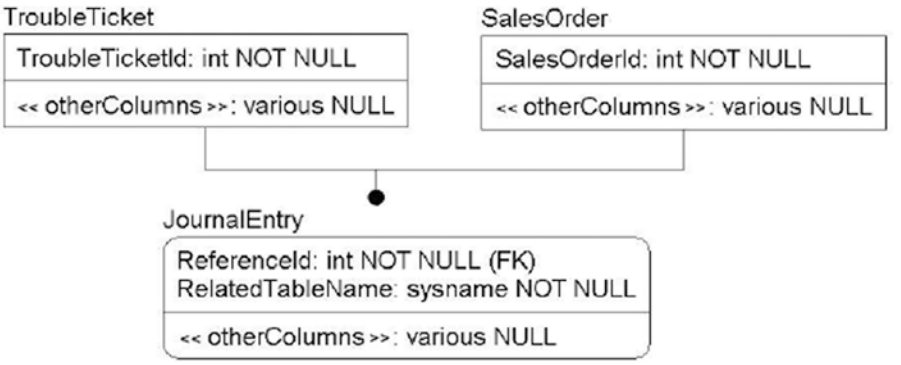


Figure 8-19. *Multiple tables related to the same key*

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 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 an 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.

WOW! eBook
www.wowebook.org

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, 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?

Extending the design, 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 8-20.

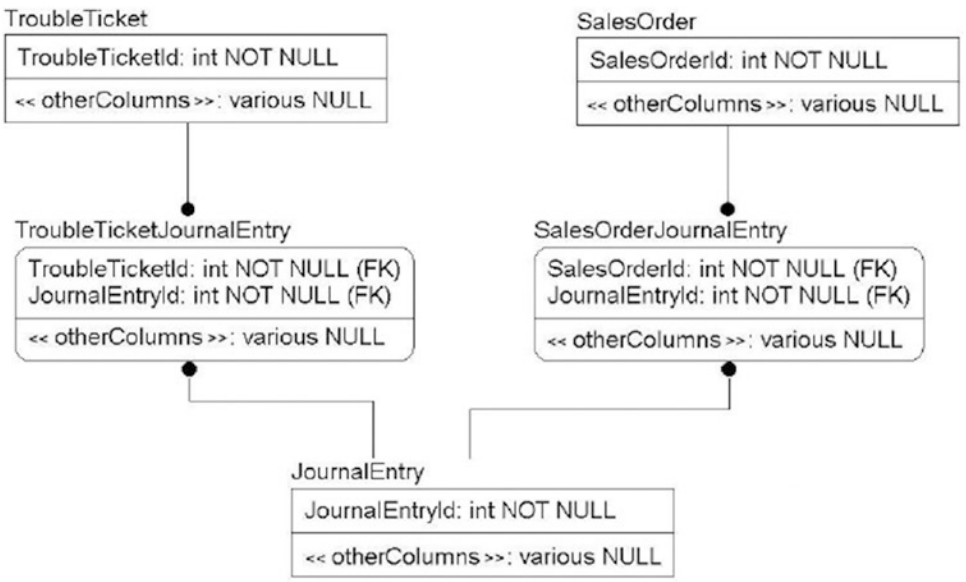


Figure 8-20. *Objects linked for maximum usability/flexibility*

WOW! eBook
www.wowebook.org

```
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, though it does make it harder to implement. 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.

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 systems I have created in my career included some column that allowed users to insert freeform text. 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 some bit of information about their data? I know I personally have tons of OneNote notebooks with lots of unstructured data.

However, if given too much leeway, or too many generic buckets for text strewn about in a database, far too often what happens is that notes become a replacement for doing actual design. The notes section becomes a replacement for things that ought to be a full-blown column. Should we have a column for special dietary restrictions? Nah, just put it in the notes column. Once the users do something once and particularly find 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." And then it costs the programmers 200 hours sorting out unstructured data.

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 I 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.

WOW! eBook
www.wowebook.org

```
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.
--------------------------------------------------------------------------------
```

This generally is not the best solution to the problem, even for a very small organization. The proper solution is almost certainly 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)
)
```

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 be jobless. Though one probably should have expected such out of a guy named Stu Pidd (obviously).

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.

WOW! eBook
www.wowebook.org

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 following are the "good" patterns we covered:

- *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 is to build your databases to be flexible enough that adding new data to the database that looks and acts like previous values does not require code changes. You do this by attempting to avoid hard-coded data that is apt to change and making columns for typical configurations.
- *Historical/temporal*: Often the user needs to be able to see their data at previous points in time as it has changed over time. I presented strategies you can use to view your data at various points in history. Using a trigger gives you a lot of control, but the new temporal extensions in SQL Server 2016 are very awesome if you can live with the (not terribly large) constraints.
- *Hierarchies*: We discussed several methods of implementing hierarchies, using simple SQL constructs to using `hierarchyId`, and introduced 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).

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.
- *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.

WOW! eBook
www.wowebook.org

- *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 on this 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.

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`.

WOW! eBook
www.wowebook.org

CHAPTER 9

Database Security and Security Patterns

To be trusted is a greater compliment than being loved.

—George MacDonald, Scottish novelist

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 and protecting data by keeping it completely inaccessible to even your users. Business needs connectivity to customers, and customers need connectivity to their data. With that having been said, security is a very important task when setting up and deploying 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, news stories report data being stolen, and the theft is inevitably due to poor security. In an earlier 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 highest profile since the last edition was the theft of a certain adultery-oriented website's records. You hopefully could not have cared less about that data, but no matter what the focus of your company's business, you store data that should never be released without the owner's permission.

Every company these days has a privacy policy, and as a database designer/programmer, meeting that policy is going to be partially your responsibility. Sometimes you will be the only person who cares about the privacy policy, and your calls for strict security will make you sound like that tinfoil hat you're wearing is not for aesthetic reasons only. There are also many laws that will govern how well you need to protect various types of data, and which data you can share with the customer, with other customers, with agencies, and even with people in the same company who work in offices in different locations. I will not even somewhat try to cover these broad privacy and legal topics, which are well beyond the scope of this book. Instead, I will cover the following topics related to securing your data so that you have the technical knowledge to meet any privacy policy or law that applies to your employer or client. If you implement these security techniques correctly, at least you will not end up being the cause of your customers' passwords, credit card numbers, and even personal proclivities being shared on the Internet for all to know.

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_9

WOW! eBook
www.wowebook.org

- *Database access*: We will cover some of the fundamentals that you need to understand about how a user gets access to a SQL Server instance and into a database.
- *Database object securables*: Once you are in the context of a database, you have a lot of built-in controls to manage what users can access. We will cover what they are, and how to use and test them.
- *Row-level security*: We will explore how to use SQL Server 2016's row-level security tools to limit access to certain rows in a table, with limited if any changes to the underlying application.
- *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 ideally are independent containers, but on occasion, you will need to access data that is not stored within the confines of the database. We will cover some of the caveats when implementing cross-database access.
- *Obfuscating data*: While the only reason to store data is to be able to read it, you want the program to be able to decode some data only when needed. This is particularly important for personally identifiable data or financial data, so we encrypt the data to keep eyes out except where allowable.
- *Auditing*: Turning on a "security camera" to watch what people are doing with data 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 deep sampling of what you need to do during database design and implementation to secure your data, but we won't cover 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 is to help you architect a security solution by showing you what is available, demonstrating some of the implementation patterns you may use, and then letting you dig in for your exact needs.

One bit of terminology clarification is important to understand. When you think of SQL Server architecture, you should think of three layers, each involved in the security (and code execution) of data:

- *Host server*: The machine (physical or virtual) that the software runs on. SQL Server primarily runs on a Windows Server platform (Linux is in beta as of this writing, and it isn't wrong to expect others to follow.). The host server can provide authentication of the identity trying to access SQL Server.
- *Instance/SQL Server*: The SQL Server installation (often referred to as just "server"). You can have multiple instances on a host server. The instance can provide authentication services of its own. SQL Server is basically an operating system of its own, using the host operating system to perform some of its tasks. Unfortunately, the term "instance" and "server" are used analogously in many contexts, likely due to historical naming that existed before the introduction of instances.
- *Database*: The container for data that the users will access. Databases can also contain authentication information.

WOW! eBook
www.wowebook.org

Additionally, if you are using an Azure SQL Database, while you will primarily deal with the database for security, you can see in the configuration of a database that the concept of a server exists, which is very similar to the server/instance for the on-premises product. Security inside the database container will behave like that of the on-premises versions that the book is mostly centered on.

Not everyone will use all of the guidelines in this chapter in their security implementations. Often, the application layer is left to implement much of the security alone, by simply 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 facilities 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.

■ **Tip** The examples in this chapter will all be of the interpreted T-SQL variety. CLR and Native objects generally follow the same patterns of security with some differences, mostly limitations on what can be done.

Database Access

In this initial section, we are going to cover a few prerequisites that we will need for the rest of this chapter on database security, starting with connecting to the server and gaining access to a database. In this section I will cover the following topics:

- *Guidelines for host server security configuration*: Some considerations to make sure your server is configured to protect against outside harm.
- *Principals and securables*: All security in SQL Server is centered on principals (loosely, logins and users) and securables (stuff that you can limit access to).
- *Connecting to the server*: With changes in SQL Server 2012, there are now multiple ways to access the server. We will cover these.
- *Impersonation*: Using the `EXECUTE AS` statement, you can "pretend" you are a different security principal to use the other user's security. It is a very important concept for testing security that we will use often in this chapter.

Guidelines for Host Server Security Configuration

It is very important to configure the host server to be as secure as possible based on how your server will be used. Very few servers these days are completely cut off from the Internet (and the scum-sucking hacker types who lurk there). As an application/data architect/programmer, I have generally only been an advisor on how to configure most of the server beyond the confines of the individual database, and deep details of on configuration is outside of the scope of this book anyhow. However, it definitely is worth a few pages to make it clear that the host server is the linchpin to the matter of securing your data. The following 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, just a list of almost universally required settings for configuring the Windows server and the SQL Server instance that we will be using to house our databases.

WOW! eBook
www.wowebook.org

- Strong passwords are applied to all accounts that can access the host server, and very strong passwords are applied to all universally known system accounts. Certainly, there are no blank passwords for any accounts! (The same will apply to accounts that can access SQL Server, both from Windows Authentication and standard accounts.)
- The host server isn't sitting unguarded on the Web, with no firewall, standard ports used for access, and/or not logging failed login attempts.
- Application passwords are secured/encrypted and put where they can be seen only by people who need to see them (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.
- Very 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 backups in whatever form you have them in, 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% secure if the hacker has virtually unlimited time; just ask the FBI or Apple).
- Your host server 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 gain access to your hard disks (note that using Transparent Data Encryption [TDE] can help in this case). This is even more important as virtualization is becoming the de facto standard. The files for the VM are a lot easier to smuggle out of the office than a 20-pound machine with disk arrays attached.
- All features that you are not using are turned off. This pertains both to the Windows Server components (if you are not using the web server services, turn them off) and to the SQL Server installations. Windows helps by not turning on everything by default, as does SQL Server. For example, remote administrator connections, Database Mail, CLR programming, and other features are all off by default. You can enable these features and others by using the `sp_configure` stored procedure.
- You have chosen proper protocols for accessing the server. Of greatest importance is to use an encrypted connection when your application transmits sensitive data (as defined by laws, privacy policies, and requirements).

The bottom line is that most of what you will do at the database level is intended to keep your mostly honest users from seeing and doing things that they shouldn't (other than some forms of encryption, at least) and isn't nearly as important as keeping the data safe from malicious outsiders. For most of your user community, you could leave all of the data unprotected and like Dorothy and her magical shoes, if they don't know what they can do, they won't take your data and go back to Kansas. So even if you just leave security to the application layer, it is essential that someone really locks down data access to allow only the people you expect to have access.

Principals and Securables

At the very core of security in SQL Server are the concepts of principals and securables. *Principals* are those identities that may be granted permission to access particular database objects, while *securables* are things 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:

WOW! eBook
www.wowebook.org

- *Windows principals*: These represent Windows user accounts or groups, authenticated using Windows security. SQL Server trusts Windows to determine who has connected to the server, and when Windows passes an identifier to SQL Server, it trusts that if the identifiers match, things are great.
- *[SQL] Server principals*: These are server-level logins or roles that are authenticated using SQL Server–based authentication, which is implemented through storage and algorithms located in the software of the SQL Server instance.
- *Database principals*: These include database users (usually mapped to Windows logins/groups), roles (groups of users and other roles to give access to en masse), and application roles (a special type of role that can be used to let an application have different rights than the user has normally).

Securables are the things to which you can control access on all parts of the instance and database and to which you can grant principals permissions. SQL Server distinguishes between three scopes at which different objects can be secured:

- *Server scoped*: Includes logins, HTTP endpoints, availability groups, 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 scoped*: Securables with database scope are objects such as schemas, users, roles, CLR assemblies, DDL triggers, and so on, which exist inside a particular database but not within a schema.
- *Schema scoped*: This group includes those objects that reside within a schema in a database, such as tables, views, and stored procedures.

These concepts will come into play in the rest of the chapter as we walk through the different ways that you will need to secure the data in the database. You can then allow, or disallow, usage of these objects to the roles that have been created. SQL Server uses three different security statements to give rights to or take away rights from each of your roles:

- `GRANT`: Allows access to an object. A principal can be granted the same right multiple times, depending on whether they are a member of any role.
- `DENY`: Denies access to an object, regardless of whether the user has been granted the privilege from any other `GRANT`.
- `REVOKE`: Essentially a `DELETE` statement for security. Removes any `GRANT` or `DENY` permission that has been applied to an object.

Typically, you'll simply use `GRANT` to give permissions to a principal to perform tasks that are specific to the principal. `DENY` is then used only in "extreme" cases, because no matter how many other times the principal has been granted privileges to an object, the principal won't have access to it while there's one `DENY`, which will often make for a confusing security scenario when you are trying to work out why user X can't access object Y.

For a right that pertains to the entire database or server, you will use syntax like

```
GRANT <privilege> TO <principal> [WITH GRANT OPTION];
```

Including `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, change settings, view metadata, and so on.

WOW! eBook
www.wowebook.org

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, use `REVOKE` the permission, which will delete the granted access:

```
REVOKE <privilege> ON <securable> FROM <principal>; --Can also be TO instead of FROM
```

If you want to prevent the principal from using the securable, no matter any role membership, you use `DENY`:

```
DENY <privilege> ON <securable> FROM <principal>; --Can also be TO instead of FROM
```

To remove `DENY`, you would again use the `REVOKE` command. 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 (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>;
```

Or you can use

```
GRANT OBJECT::<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 `GRANT` on a schema securable, the syntax is

```
GRANT <privilege> ON SCHEMA::<schema securable> TO <database principal>;
```

Connecting to the Server

Before we finally get to 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 normal way and is basically as follows: A login principal is defined that allows a principal to access the server using Windows credentials (Windows Authentication), a login that is managed in the SQL Server instance (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.

The additional method in SQL Server 2012 uses the concept of a contained database (CDB). I will cover the broader picture and a bit of the management of CDBs 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 because it is, from a coding standpoint, largely a security question. Contained databases in SQL Server 2012 are the initial start of making databases essentially stand-alone containers that can be moved from server to server with little effort, and into an Azure Database as well.

WOW! eBook
www.wowebook.org

In this section, I will provide two examples of connecting to the server:

- Using the classic approach of a login and database user to connect to the server first, and then the database
- Accessing the database directly using the containment model

Connecting to the Server Using a Login and a Database 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 [DomainName\Louis] FROM WINDOWS --square brackets required for WinAuth login
    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 (the name of which I will replace with `DomainName`), I have a user named `Louis`. 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. If you are using Azure SQL DB, you can use Azure Active Directory Authentication to do very much the same thing (azure.microsoft.com/en-us/documentation/articles/sql-database-aad-authentication/).

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 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, and `'password'` is not likely to pass, even on a simple machine you have created just for trying out this code, so pick something that will meet your requirements. Typical password requirements should be similar to

- Should be pretty long, not just one character from each of the following bullets (for tips, see, e.g., www.infoworld.com/article/2655121/security/password-size-does-matter.html)
- Should contain uppercase symbols (A-Z)
- Should contain lowercase symbols (a-z)
- Should contain numbers (0-9)
- Should contain at least one symbol from this list: _, @, *, ^, %, !, #, $, or &

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 they will tape a list of passwords to 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 complex passwords and (ideally) change them often.

WOW! eBook
www.wowebook.org

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 objects 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` server role, 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 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 [DomainUser\Louis];
```

■ **Tip** Members of the `sysadmin` role basically bypass almost all rights checks on the server and are allowed to do anything ("almost all" because they will be subject to row-level security, data masking, and anything you code in a trigger). 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, if Fred works in support, you may want to give him read-only access to a server (without rights to change anything). First off, say you want Fred to be able to run DMVs (Dynamic Management Views, which are sometimes functions) to see the state of the server. You would grant the `Fred` user `VIEW SERVER STATE` permission using

```
GRANT VIEW SERVER STATE to Fred;
```

As of SQL Server 2012, you could create user-defined server roles. For example, say you want to set up a role to let the user view the server settings and data. You could give the following rights:

- `VIEW SERVER STATE`: Access DMVs (previously mentioned)
- `VIEW ANY DATABASE`: See the structure of all databases
- `CONNECT ANY DATABASE`: Connect to any existing and future database
- `SELECT ALL USER SECURABLES`: View all data in databases the login can connect to

To create a server role for these items, you could use

```
CREATE SERVER ROLE SupportViewServer;
```

Grant the role the rights desired as follows:

```
GRANT  VIEW SERVER STATE to SupportViewServer; --run DMVs
GRANT  VIEW ANY DATABASE to SupportViewServer; --see any database
GRANT  CONNECT ANY DATABASE to SupportViewServer; --set context to any database
GRANT  SELECT ALL USER SECURABLES to SupportViewServer; --see any data in databases
```

And add the login to the server role:

```
ALTER SERVER ROLE SupportViewServer ADD MEMBER Fred;
```


WOW! eBook
www.wowebook.org

Once you have created your login, the next step is to access a database (unless you used `sysadmin`, in which case you have unfettered access to everything on the server). For the first examples in this chapter, create a database called `ClassicSecurityExample`, as shown next. (For the remainder of this chapter, I will expect that you are 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 I specify a different user in the text.)

```
CREATE DATABASE ClassicSecurityExample;
```

Next, create another login that uses SQL Server Authentication. Most logins we will create in the book will be SQL Server Authentication to make it easier to test. 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;
```

Log in using the user in Management Studio into a query window, as shown in Figure 9-1.

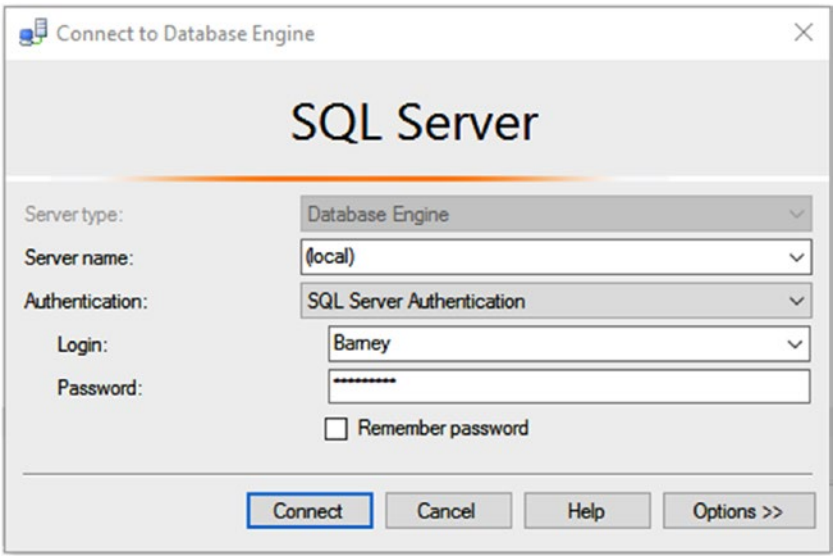


Figure 9-1. *Logging in using test user*

Next, try to execute a `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.
```


WOW! eBook
www.wowebook.org

Your database context will remain in tempdb, since this is the default database you set up for the user. Going back to the window where you are in the sysadmin user context, you 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 (back as the sysadmin user, naturally). The guest user is a built-in user that every database has. It equates to "anyone who connects," basically.

```
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, 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 like SELECT 'hi'; 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, create a user in the database linked to the login and grant it the right to connect:

```
USE ClassicSecurityExample;
GO
CREATE USER BarneyUser FROM LOGIN Barney;
GO
GRANT CONNECT to BarneyUser;
```

Going back to the query window in the context of Barney, you will find that you can connect to the database and, using a few system functions, 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 something like (depending on what you used to log in to the server):

```
server_principal_name   database_principal_name
----------------------- ------------------------------
DOMAINName\Louis        dbo
```


WOW! eBook
www.wowebook.org

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 as recently as SQL Server 2008, the advice would have been strongly against using any sort of virtualization technology with SQL Server, and now our company runs everything on virtualized Windows machines. 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.

Another paradigm shift is cloud computing. Azure DB databases behave like self-contained database servers. When you connect to them, you see a version of a `master` database, and a database container. Logins do not reside at the server level, but are contained in the database. In the on-premises product, we can do the same thing using the "containment" model. The idea behind containment is that everything your database needs (jobs, ETL, `tempdb` objects, etc.) will begin to be a part of the database directly. Where applicable, I will note some of the places where contained database security is different from the classic model, which is mostly in the context of accessing external objects.

Your first step is to create a new database in which you will set `containment = partial`. For SQL Server 2012 and later, there are two models: `OFF`, which I am referring to as the classic model, and `partial`, which will give you a few benefits (like temporary object collation defaulting to the partially contained databases rather than the server). Not much has changed in containment since 2012, but later versions of SQL Server may 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 discussed in Chapter 8, this means a security point that is going to be turned off by default. Hence, the first thing you 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 can set the containment properties in the `CREATE DATABASE` statement:

```
CREATE DATABASE ContainedDBSecurityExample CONTAINMENT = PARTIAL;
```

Or you can set it using an `ALTER DATABASE` statement:

```
-- set the contained database to be partial
ALTER DATABASE ContainedDBSecurityExample SET CONTAINMENT = PARTIAL;
```

Next, you will create a user, which in this context is referred to as a "contained user." Contained users are basically a hybrid of login and user, 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 at least 11 variations of the `CREATE USER` syntax, so you should check it out if you need a different sort of user!

The first case you 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 you set earlier), or you will get an error telling you that you can only create a user with a password in a contained database.

WOW! eBook
www.wowebook.org

```
USE ContainedDBSecurityExample;
GO
CREATE USER WilmaContainedUser WITH PASSWORD = 'p@ssword1';
```

You can also create a Windows Authentication user in the following manner (it could be a role as well) 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 [DOMAIN\Louis];
```

Since that user already has a login, I 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, during the testing phase, we will be using SQL Server Authentication to make the process easier.

Next, connect to the database in SSMS using the contained user you previously created named `WilmaContainedUser` with password `p@ssword1`. To do this, specify the server name, choose SQL Server Authentication, and set the username and password, as shown in Figure 9-2.

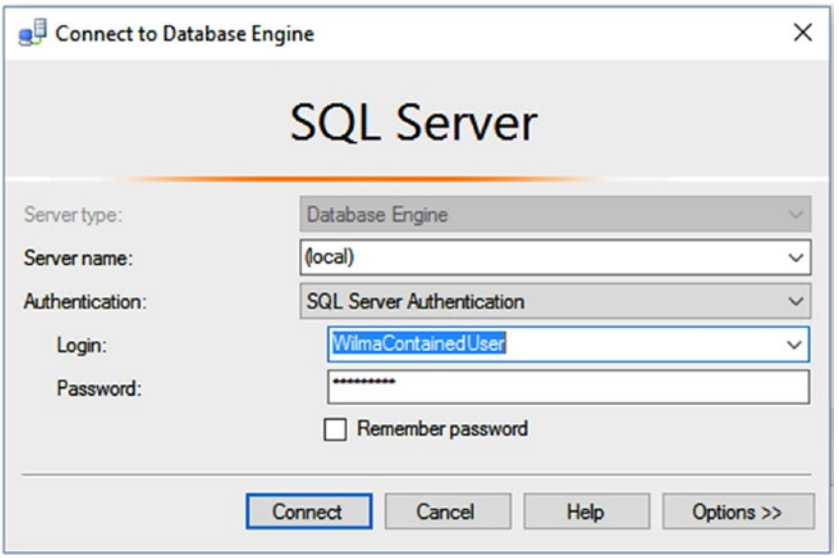


Figure 9-2. *Demonstrating logging in to a contained user*

Next, click the Options button. Go to the Connection Properties tab, and enter the name of the contained database as shown in Figure 9-3.

WOW! eBook
www.wowebook.org

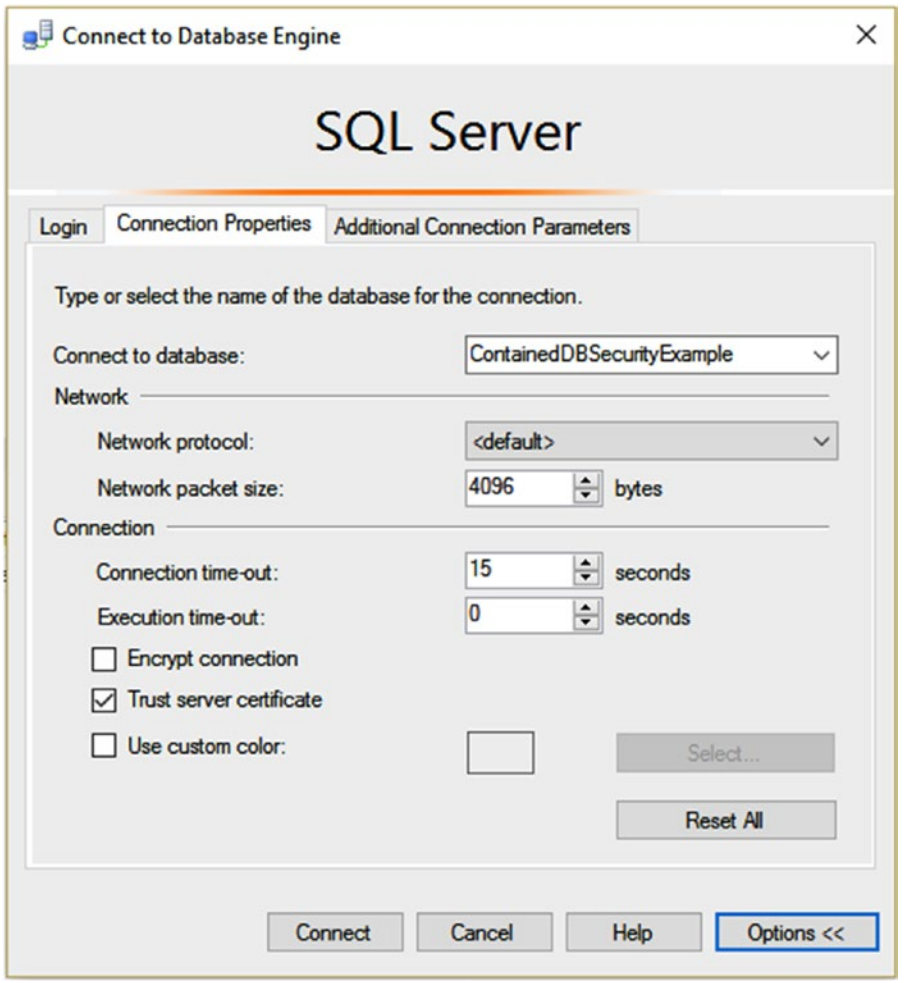


Figure 9-3. *Enter the name of the database in the blank.*

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.

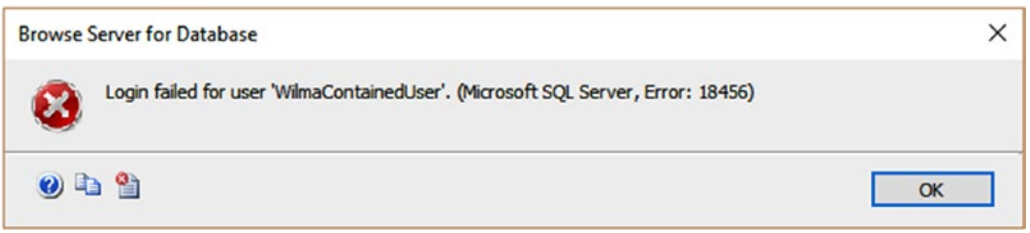


Figure 9-4. *Error trying to browse for name of contained database*

WOW! eBook
www.wowebook.org

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.

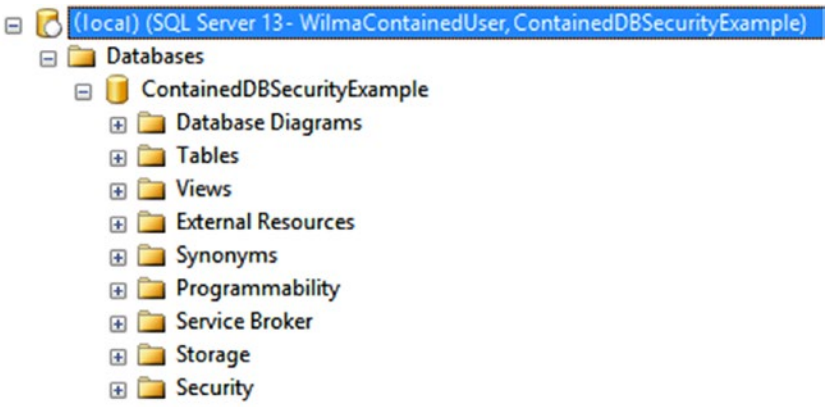


Figure 9-5. *Contained database in Object Explorer in SSMS*

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 classic security model covered in the previous section.

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;
```

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;
```


WOW! eBook
www.wowebook.org

This will fail. In SQL Server 2012, you get a confusing error message:

```
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.
```

But in SQL Server 2016, the error message is very clear:

```
Msg 12809, Level 16, State 1, Line 103
You must remove all users with password before setting the containment property to NONE.
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   ContainedDBSecurityExample.sys.database_principals --3 part name since you are outside
                                                          --of db to make this change.
WHERE  authentication_type_desc = 'DATABASE';
```

In our example, 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. After some code has been migrated to production, it is not an uncommon occurrence to get a call from clients who claim 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. (There is a server-scoped permission `IMPERSONATE ANY LOGIN` that was added to SQL Server 2014 to make this easier for a semi-`sysadmin` type to do.)

To demonstrate security in a reasonable manner on a single SQL Server connection, I will use the `EXECUTE AS` command to impersonate different security principals, both database and server.

WOW! eBook
www.wowebook.org

■ **Note** Prior to 2005, the command to impersonate was `SETUSER`. Some use of that command in legacy code is still possible, as it still works. `SETUSER` is limited in comparison to `EXECUTE AS`, but is similar.

As an example of how powerful impersonation can be, 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.

In this example, I use a SQL Server Authentication 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 (which is part of what makes them less secure for production use!). Then, I 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 SlateSystemAdmin WITH PASSWORD = 'tooHardToEnterAndNoOneKnowsIt',
CHECK_POLICY=OFF;

ALTER SERVER ROLE sysadmin ADD MEMBER SlateSystemAdmin;
```

Then, I create a regular login and give rights to impersonate the `system_admin` user:

```
CREATE LOGIN Slate with PASSWORD = 'reasonable', DEFAULT_DATABASE=tempdb,CHECK_POLICY=OFF;


--Must execute in master Database
GRANT IMPERSONATE ON LOGIN::SlateSystemAdmin TO Slate;
```

■ **Caution** You probably do 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.

I log in as `Slate` 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;
```


WOW! eBook
www.wowebook.org

The following error is raised:

```
Msg 916, Level 14, State 1, Line 1
The server principal "Slate" is not able to access the database "ClassicSecurityExample"
under the current security context.
```

Now, I change security context to the `system_admin` level user (note that you cannot use `EXECUTE AS LOGIN` when you are in the context of a contained database user):

```
EXECUTE AS LOGIN = 'SlateSystemAdmin';
```

I now have control of the server in that window as the `system_admin` user! To look at the security context, I 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           SlateSystemAdmin     Slate
```

The columns mean the following:

- `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, I execute the following code:

```
REVERT; --go back to previous security context
```

I 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.
```


WOW! eBook
www.wowebook.org

I started in tempdb, so I use the following code:

```
USE tempdb;

REVERT;
SELECT USER AS [user], SYSTEM_USER AS [system_user],
       ORIGINAL_LOGIN() AS [original_login];
```

This now returns the following result:

```
user          system_user          original_login
------------- -------------------- ------------------------
guest         Slate                Slate
```

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.

■ **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 role 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 Object 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 this section, we are going to cover the different ways you can now use the database principals you have created.

I'll cover the basics of database permissions for a foundation of best practices. Taken to the extreme, the set of things considered securable is extensively large, especially at the server level, but over 90% of security activity 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. Everything else is very similar in any case.

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 other principal we will start to use is a 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

WOW! eBook
www.wowebook.org

(and all environments in between) and helps avoid users who end up with god-like powers from getting one permission here, another there, and so on. The roles will be the same in all areas; allowing most of your security code to look the same (and hence be checked into source control and testable) and the different users who are associated with the roles are then different in production, test, and so on can be different, if perhaps similar to make things clearer.

I'll cover the following topics, which revolve around giving users permissions to use securables:

- *Grantable permissions*: You'll learn about 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

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). There are also rights you can use to allow a user to make changes to a table (`ALTER`) or do anything that is available with `CONTROL`. If you want to give all of the listed permissions to a principal, use `ALL` instead of the permission name.

Table 9-1. *Database Objects and Permissions*

Object Type	Permission Type
Tables, views	`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`
Columns (view and table)	`SELECT`, `INSERT`, `UPDATE`, `DELETE`
Scalar functions	`EXECUTE`, `REFERENCES`
Table-valued functions (not all will apply in all functions)	`SELECT`, `UPDATE`, `DELETE`, `INSERT`, `REFERENCES`
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 as it is not used very often. 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 object owned by another via a foreign key. For 99.5% of databases, all objects should be owned by the same user. For the other .49% of them, the objects might be owned by different users, but you would probably not want to implement foreign keys between the tables. Hence we will largely ignore the `REFERENCES` right.

As briefly mentioned earlier in this chapter for server and database permissions, you will use one of the three different statements to give rights to or take away rights from each of your roles:

- `GRANT`: Gives a right
- `DENY`: Disallows access to an object, regardless of any other associated grants
- `REVOKE`: Deletes a previously applied `GRANT` or `DENY` permission

WOW! eBook
www.wowebook.org

To see the user's rights in a 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;
```

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.

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 (you can also grant the `REFERENCES` right, which allows the user to reference the object in a foreign key, but this is rare). 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.

■ **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, I will create a new table, and 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, I 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;
```


WOW! eBook
www.wowebook.org

■ **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 when you restore a database that has existing users but no login on the server.

I 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 154
The INSERT permission was denied on the object 'TableExample', database
'ClassicSecurityExample', schema 'TestPerms'.
```

Now, I 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, I try to execute the insert statement again as Tony; I 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 168
The SELECT permission was denied on the object 'TableExample', database
'ClassicSecurityExample', schema 'TestPerms'.
```

Now, I 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;
```


WOW! eBook
www.wowebook.org

Now that Tony has rights, I can successfully run the following:

```
EXECUTE AS USER = 'Tony';

SELECT TableExampleId, Value
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 statement permission types 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 present 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 NOT NULL IDENTITY CONSTRAINT PKProduct PRIMARY KEY,
    ProductCode varchar(10) NOT NULL CONSTRAINT AKProduct_ProductCode UNIQUE,
    Description varchar(20) NOT NULL,
    UnitPrice   decimal(10,4) NOT NULL,
    ActualCost  decimal(10,4) NOT NULL
);
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:

WOW! eBook
www.wowebook.org

```
GRANT SELECT on Products.Product to employee,manager;
DENY SELECT on Products.Product (ActualCost) to employee;
```

To test our security, we impersonate the user manager:

```
EXECUTE AS USER = 'manager';
SELECT  *
FROM    Products.Product;
```

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** You may be 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 can save a trip to an occupational therapist for pain from too much typing, even using IntelliSense.)

The user manager worked fine; what about the user 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;
```


WOW! eBook
www.wowebook.org

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
```

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 (this is an advanced user!) on the `Product` table (which has 87 columns instead of the 5 I've generously mocked up for your learning ease), I have to type out all the columns? And if I use some form of tooling, I have to remember what columns I don't have access to?"

This is why, for the most part, column-level security is rarely used as a primary security mechanism. 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 determining who to grant rights to. 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 the following 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 giving rights 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.

WOW! eBook
www.wowebook.org

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.
- `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 objects that you may not want users, or even the application, 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 wanted 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 execute 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 role use to utility users, perhaps an admin type, or an ETL program's access that will not be doing any ad hoc queries that could be in error (after considerable testing, naturally).

User-Defined Database Roles

Just like at the server level, 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.

WOW! eBook
www.wowebook.org

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.
- `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.

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` role 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.

I won't do it in this example, but you may then want to create a role titled `TypeOfManager` that is a member of all three roles. This would allow you to define the roles in the company to the level of employee A is a member of `TypeOfManager`, and it covers all that you need. A `SecondTypeOfManager` may have several of the same role memberships, but also have a few more. Crafting a security configuration is complex, but once it is created and tested, it is a wonderful thing.

Programmatically, you can determine some basic information about a user's security information in the database with the following functions:

- `IS_MEMBER('<role>')`: Tells you whether the current user is the member of a given role. This is useful for building security-based views. You can also pass in a Windows Group to see if the user is a member of that group.
- `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.

WOW! eBook
www.wowebook.org

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 of the role (0 means not a member, and NULL means the role doesn't exist). 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.

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 add

```
CREATE ROLE HRWorkers;

ALTER ROLE HRWorkers ADD MEMBER Julie;
ALTER ROLE HRWorkers ADD MEMBER Rie;
```

■ **Tip** ALTER ROLE was new to SQL Server 2012. It replaced sp_addrolemember, which has been deprecated, so you should make a habit of using ALTER ROLE instead as you write code.

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 NOT NULL CONSTRAINT PKEmployeeSalary PRIMARY KEY,
    SalaryAmount decimal(12,2) NOT NULL
);
GRANT SELECT ON Payroll.EmployeeSalary to HRWorkers;
```


WOW! eBook
www.wowebook.org

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 253
The SELECT permission was denied on the object 'EmployeeSalary', database
'ClassicSecurityExample', schema 'Payroll'.
```

However, change over to Julie

```
REVERT;
EXECUTE AS USER = 'Julie';

SELECT *
FROM   Payroll.EmployeeSalary;
```

and you find that Julie can view the data of tables in the Payroll schema because Julie is 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 UNIONed. 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 has had rights to the EmployeeSalary table denied:

```
REVERT;
DENY SELECT ON Payroll.EmployeeSalary TO Rie;
```

If Rie now tried to select from the table

```
EXECUTE AS USER = 'Rie';
SELECT *
FROM   Payroll.EmployeeSalary;
```

access would be denied:

```
Msg 229, Level 14, State 5, Line 2
The SELECT permission was denied on the object 'EmployeeSalary', database
'ClassicSecurityExample', schema 'Payroll'.
```


WOW! eBook
www.wowebook.org

This denial of access is true even though Rie 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,
        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;
REVERT ; --so you will be back to sysadmin rights for next code
```

This returns

```
tableName               allowSelect allowInsert
----------------------- ----------- -----------
Payroll.EmployeeSalary  1           0
```

User Julie has rights to see only one of the tables we have created, and 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.

■ **Tip** HAS_PERMS_BY_NAME can be used to see if a user has rights to a column as well. If you choose to use column-level security, you could use this to generate SELECT statements for your user.

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.

WOW! eBook
www.wowebook.org

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, 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 NOT NULL IDENTITY(1,1) CONSTRAINT PKBobCan PRIMARY KEY,
    Value varchar(10) NOT NULL
);
CREATE TABLE TestPerms.AppCan
(
    AppCanId int NOT NULL IDENTITY(1,1) CONSTRAINT PKAppCan PRIMARY KEY,
    Value varchar(10) NOT NULL
);
```

Now, create the user Bob to correspond to the BobCan table:

```
CREATE USER Bob WITHOUT LOGIN;
```

Next, give Bob SELECT rights to the BobCan 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;
```

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 OLE DB client. For more security, use an encrypted connection.

Next, set the user you're working as to Bob and try to retrieve data from the BobCan table:

```
EXECUTE AS USER = 'Bob';
SELECT * FROM TestPerms.BobCan;
```

It works with no error:

```
BobCanId    Value
----------- ----------
```


WOW! eBook
www.wowebook.org

However, try retrieving data from the AppCan table:

```
SELECT * FROM TestPerms.AppCan;
```

The following error is returned:

```
Msg 229, Level 14, State 5, Line 315
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 in the context of user Bob, use the sp_setapprole procedure to change the security context of the user to the application role, and the security is reversed:

```
EXECUTE sp_setapprole 'AppCan_application', '39292LjAsll2$3';
GO
SELECT * FROM TestPerms.BobCan;
```

This returns the following error:

```
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;
```

This returns the following result:

```
userName
--------------------
AppCan_application
```


WOW! eBook
www.wowebook.org

Once you've executed `sp_setapprole`, the security stays as this role until you disconnect from the SQL Server 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
```

This returns the following results:

```
Cookie
------------------------------------------------------------------------------------------
0x39881A28E9FB46A0A002ABA31C11B7F4C149D8CB2BCF99B7863FFF729E2BE48F13C0F83BAD62CF0B221A863B83
beforeUnsetApprole
------------------------------------------------------------------------------AppCan_application
afterUnsetApprole
------------------------------------------------------------------------------------------
dbo
```

The cookie is an interesting value, much larger than a GUID—it is declared as `varbinary(8000)` in Books Online, even though the current value is considerably less wide. 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 the first step. 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

WOW! eBook
www.wowebook.org

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
- 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 `WideWorldImporters` database, use the following query of the `sys.objects` catalog view (which reflects schema-scoped objects):

```
USE WideWorldImporters; --or whatever name you have given it
GO
SELECT  SCHEMA_NAME(schema_id) AS schema_name, type_desc, COUNT(*)
FROM    sys.objects
WHERE   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  SCHEMA_NAME(schema_id), type_desc
ORDER BY schema_name;
GO
USE ClassicSecurityExample;
```

This query shows how many of each object can be found in the version of the `WideWorldImporters` database in each schema I have on your development machine. As shown in syntax 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 can execute all 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;
```


WOW! eBook
www.wowebook.org

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 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 access to.

Row-Level Security

So far, we have discussed only methods of securing data at the object level. If you have access to Table A, you have access to all of the rows of Table A. In this section, we will look at several methods of securing the rows of the table, letting a user see only part of the rows in the table. In SQL Server 2016, there is a new feature called Row-Level Security, but row-level security is something that I have needed for 20+ years, and the methods previously used are still interesting for now and years to follow.

We will look at three methods for performing row-level security, including the new feature:

- *Specific-purpose views*: Creating a view that represents a specific subset of rows in a table.
- *Row-Level Security feature*: There are several ways this feature will work, allowing you to show a subset to a user with very few changes to your application architecture.
- *Data-driven security views*: Basically the precursor to the Row-Level Security feature.

For the examples in this section, we will continue using the Products.Product table. To remind you of the structure, it is repeated here:

```
CREATE TABLE Products.Product
(
    ProductId   int NOT NULL IDENTITY CONSTRAINT PKProduct PRIMARY KEY,
    ProductCode varchar(10) NOT NULL CONSTRAINT AKProduct_ProductCode UNIQUE,
    Description varchar(20) NOT NULL,
```


WOW! eBook
www.wowebook.org

```
    UnitPrice   decimal(10,4) NOT NULL,
    ActualCost  decimal(10,4) NOT NULL
);
```

To keep our examples fairly simple, I am going to add a categorization column that our security examples will use to partition on and set the values to our usual generally silly values:

```
ALTER TABLE Products.Product
   ADD ProductType varchar(20) NOT NULL
                       CONSTRAINT DFLTProduct_ProductType DEFAULT ('not set');
GO
UPDATE Products.Product
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 contents that we will work with, giving a database principal access to a given product type:

ProductId	ProductCode	Description	UnitPrice	ActualCost	ProductType
1	widget12	widget number 12	10.5000	8.5000	widget
2	snurf98	Snurfulator	99.9900	2.5000	snurf

One concept that we need to introduce at this point is *ownership chaining*. 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

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.

We will see more of ownership chaining later in the chapter when we discuss stored procedures, but it is important at this point because it affects row-level security. Say we have table `S.T`, and a view that is defined as `CREATE VIEW S.V AS SELECT C FROM S.T;`. If a user has rights to view `S.V`, as long as the tables referenced in the view (in this case `S.T`) are owned by the same database principal, no rights are needed on the objects used. This allows us to use views (and later functions and procedures) to control access to objects situationally. In this section we will use views to limit access to a table, and the user will have rights to the view only.

WOW! eBook
www.wowebook.org

Using Specific-Purpose Views to Provide Row-Level Security

The absolute simplest version of row-level security is just building views to partition the data in some form, such as all products of a given type, employees in Division X, etc. Whether or not this makes sense for your usage is really up to how the data will be used. Usually this is not going to be flexible enough for your needs, but it really depends on how deep the domain is you are working with.

As an example, our `products` table is a case where, assuming the range of values supported by `ProductType` isn't large, simple row-level security may make sense. So let's configure a view that allows you to just see one particular type of product. You can build the following view:

```
CREATE VIEW Products.WidgetProduct
AS
SELECT ProductId, ProductCode, Description, UnitPrice, ActualCost, ProductType
FROM   Products.Product
WHERE  ProductType = 'widget'
WITH   CHECK OPTION; --This prevents the user from INSERTING/UPDATING data that would not
                     --match the view's criteria
```

Note a few important things about this view. First, I named it like it was a table. Usually when we use a view for row-level security, we want it to seem like a table to the user. Second, I included the `ProductType` column. We would need that column if we wanted to allow the user to `INSERT` new rows into the table using the view. Using an `INSTEAD OF` trigger, we could get around the need to include missing columns, if desired, but this is a lot easier to build. Finally, the `CHECK OPTION` says that if someone inserts data using the view, the `ProductType` would have to be set to `'widget'` or it would fail.

To test this, I will create a user `chrissy` who I will give access to the view:

```
CREATE USER chrissy WITHOUT LOGIN;
GO
GRANT SELECT ON Products.WidgetProduct TO chrissy;
```

Now the user comes in, does their query, and only sees the subset of rows:

```
EXECUTE AS USER = 'chrissy';
SELECT *
FROM   Products.WidgetProduct;
```

This returns the following result:

```
ProductId   ProductCode Description          UnitPrice    ActualCost    ProductType
----------- ----------- -------------------- ------------ ------------- ------------------
1           widget12    widget number 12     10.5000      8.5000        widget
```

However, just to make sure, try to `SELECT` rows from the `Product` table:

```
SELECT *
FROM   Products.Product;
GO
REVERT;
```


WOW! eBook
www.wowebook.org

This returns the following error message:

```
Msg 229, Level 14, State 5, Line 423
The SELECT permission was denied on the object 'Product', database
'ClassicSecurityExample', schema 'Products'.
```

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 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. I won't demonstrate it in the text, but there is code in the downloads that shows modification rights.

In the next step, let's build in some more flexible security to a view, allowing everyone to see any ProductType other than 'snurf', and only members of a certain group ('snurfViewer', naturally) to view the snurf products:

```
CREATE VIEW Products.ProductSelective
AS
SELECT ProductId, ProductCode, Description, UnitPrice, ActualCost, ProductType
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;
```

I called this ProductSelective, but here is where the naming gets tricky. If this is a reporting view, I might call it ProductView. In this case, I am treating it like it is meant to be used like a normal table. I will set the view to be usable by anyone who has access to the database (good for demo, perhaps less good for production work!). Note that you are not limited to such a simple predicate for the view, but the more complex the need, the more taxing it might be on performance.

```
GRANT SELECT ON Products.ProductSelective to public;
```

Next, add a new snurfViewer role. Note that you don't add this user to the group yet; you'll do that later in the example.

```
CREATE ROLE snurfViewer;
```

Then, change security context to chrissy and select from the view:

```
EXECUTE AS USER = 'chrissy';
SELECT * from Products.ProductSelective;
REVERT;
```

This returns the one row to which chrissy has access:

```
ProductId   ProductCode Description          UnitPrice    ActualCost    ProductType
----------- ----------- -------------------- ------------ ------------- ------------------
1           widget12    widget number 12     10.5000      8.5000        widget
```


WOW! eBook
www.wowebook.org

Next, 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 technique of using specific, hard-coded views is the simplest to configure, but lacks some level of elegance, and ends up being very difficult to manage. The next methods will take this concept to a deeper level.

Using the Row-Level Security Feature

Row-Level Security (RLS) is a new feature in SQL Server 2016 that gives you highly configurable methods of securing rows in a table in a manner that may not require any changes to an application. Basically, you configure one or more predicate functions (a simple table-valued user-defined function that returns either 1 or nothing) to indicate that a user can or cannot see or modify a row in the table. I will note too that, depending on how much access a user has to a server, there are a few side-channel ways a user might be able to infer the data in the table. RLS is designed to block these side channels. For example, when using RLS, a user will not be able to see the plan of the query (since that could show estimated row counts). For more information, check the Row-Level Security page in Books Online: https://msdn.microsoft.com/en-us/library/dn765131.aspx.

In the following example, we will continue with the same scenario from the previous section, but instead of creating a view, we will apply the security to the table itself (you can apply Row-Level Security to a schema-bound view just like you can a full table).

To start with, we will reset things from the previous scenario by giving a user rights to read and modify the rows of the Products.Product table. In order to make security easier to manage, we will put all of the row-level security objects in a separate schema. Unlike the separate schemas we created for the temporal feature in Chapter 8, no users will need access to the objects in this schema other than when we are testing the functionality, so we will just create one schema:

```
CREATE SCHEMA RowLevelSecurity;
```

Next we will create a user-defined function that provides the exact same security configuration as we used in the previous section for a view. Every user can see all rows except the ones of product type 'snurf', and the db_owner role members can see everything. The database principal will also need SELECT rights to the table we will apply this to.

WOW! eBook
www.wowebook.org

```
CREATE FUNCTION RowLevelSecurity.Products_Product$SecurityPredicate
                                                    (@ProductType AS varchar(20))
    RETURNS TABLE
WITH SCHEMABINDING --not required, but a good idea nevertheless
AS
    RETURN (SELECT 1 AS Products_Product$SecurityPredicate
            WHERE  @ProductType <> 'snurf'
                           OR  (IS_MEMBER('snurfViewer') = 1)
                           OR (IS_MEMBER('db_owner') = 1));
```

Unlike with most security features in SQL Server, sa and dbo users are subject to the row-level security predicates that are configured. Most of the time, for supportability, you will want to not make that the case, but it can be useful. Another difference between this and other security features is how they behave in coded objects. By default, even when using the base table in an object like a procedure, rows will be filtered by row-level security. I will show this later in the chapter when I discuss impersonation in objects.

While you will not generally want to grant rights to the security function, when testing it is useful to make sure how the function will behave on its own. So we start by creating and granting a new user rights to select from the function:

```
CREATE USER valerie WITHOUT LOGIN;
GO
GRANT SELECT ON RowLevelSecurity.Products_Product$SecurityPredicate TO valerie;
```

Now we test to see what the output will be:

```
EXECUTE AS USER = 'valerie';
GO
SELECT 'snurf' AS ProductType,*
FROM   rowLevelSecurity.Products_Product$SecurityPredicate('snurf')
UNION ALL
SELECT 'widget' AS ProductType,*
FROM   rowLevelSecurity.Products_Product$SecurityPredicate('widget');

REVERT;
```

This returns just the one row for the widget type of row:

```
ProductType Products_Product$SecurityPredicate
----------- ---------------------------------
widget      1
```

When we feel confident that the function is correct, we remove user access from the row-level security function:

```
REVOKE SELECT ON RowLevelSecurity.Products_Product$SecurityPredicate TO valerie;
```

Now that we have the function created, we apply it using the CREATE SECURITY POLICY statement. You can create two kinds of predicates: FILTER, which filters out viewing rows, and BLOCK, which disallows certain operations. We will start with a simple filter. The user will need to have access to SELECT from the table already, and then this security policy will take away access to certain rows within.

WOW! eBook
www.wowebook.org

The following command adds a filter predicate that will pass in the ProductType value to the function and see if it returns anything. Because this is row-level security, this function will be executed once per row, so keep the function as simple as possible to limit the overhead. Since it is a simple table-valued function, it is much more difficult to make things slow, but it is always possible.

```
CREATE SECURITY POLICY RowLevelSecurity.Products_Product_SecurityPolicy
    ADD FILTER PREDICATE rowLevelSecurity.Products_Product$SecurityPredicate(ProductType)
    ON Products.Product
    WITH (STATE = ON, SCHEMABINDING = ON);
```

State=ON turns on the policy to be checked. SCHEMABINDING=ON says that the predicate functions must all be schema bound. Not having the function schema bound would allow you to access objects in a different database if you needed to, but generally it is considered a bad idea. You can only have one enabled filter predicate per table. If you try to put another in the same policy, you get either message, depending on if you try to create two in the same policy, or different policies. For example:

```
CREATE SECURITY POLICY rowLevelSecurity.Products_Product_SecurityPolicy2
    ADD FILTER PREDICATE rowLevelSecurity.Products_Product$SecurityPredicate(ProductType)
    ON Products.Product
    WITH (STATE = ON, SCHEMABINDING= ON);
```

This will cause the following message:

```
Msg 33264, Level 16, State 1, Line 607
The security policy 'rowLevelSecurity.Products_Product_SecurityPolicy2' cannot be
enabled with a predicate on table 'Products.Product'. Table 'Products.Product' is
already referenced by the enabled security policy 'rowLevelSecurity.Products_Product_
SecurityPolicy'
```

We start by giving the user all rights on the table, to enable the rest of the demos. We will limit the user's ability to perform these tasks using row-level security.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON Products.Product TO valerie;
```

Now, with the policy in place, let's see what the user can see:

```
EXECUTE AS USER = 'valerie';

SELECT *
FROM   Products.Product;

REVERT;
```

```
ProductId   ProductCode Description          UnitPrice    ActualCost    ProductType
----------- ----------- -------------------- ------------ ------------- ------------------
1           widget12    widget number 12     10.5000      8.5000        widget
```


WOW! eBook
www.wowebook.org

We have filtered the rows down to the non-snurf rows. The filter works not just for SELECT, but for DELETE and UPDATE too. For example, we know that there are snurf rows, but executing as user valerie, they can't be modified:

```
EXECUTE AS USER = 'valerie';

DELETE Products.Product
WHERE  ProductType = 'snurf';

REVERT;

--back as dbo user
SELECT *
FROM   Products.Product
WHERE  ProductType = 'snurf';
```

```
ProductId   ProductCode Description          UnitPrice    ActualCost    ProductType
----------- ----------- -------------------- ------------ ------------- ------------------
2           snurf98     Snurfulator          99.9900      2.5000        snurf
```

However, just because we can't see a row doesn't mean we can't create a row of that type:

```
EXECUTE AS USER = 'valerie';

INSERT INTO Products.Product (ProductCode, Description, UnitPrice, ActualCost,ProductType)
VALUES  ('Test' , 'Test' , 100 , 100  , 'snurf');

SELECT *
FROM   Products.Product
WHERE  ProductType = 'snurf';

REVERT;

SELECT *
FROM   Products.Product
WHERE  ProductType = 'snurf';
```

We get back:

```
ProductId   ProductCode Description          UnitPrice    ActualCost    ProductType
----------- ----------- -------------------- ------------ ------------- ------------------
2           snurf98     Snurfulator          99.9900      2.5000        snurf

ProductId   ProductCode Description          UnitPrice    ActualCost    ProductType
----------- ----------- -------------------- ------------ ------------- -----------------
2           snurf98     Snurfulator          99.9900      2.5000        snurf
5           Test        Test                 100.0000     100.0000      snurf
```


WOW! eBook
www.wowebook.org

To block actions from occurring, we will use a BLOCK predicate. There are two block types:

- AFTER: If the row does not match the security predicate after the operation, it will fail. So if you can see A, but not B, you could not change A to B. But you could change B to A, if you could see it.
- BEFORE: If the row does not match the security predicate before the operation, you cannot perform the operation. So if you can modify A, but not B, you could update A to B, but no longer be able to modify the row again.

We are going to set one of the seemingly obvious set of row-level security predicates that one might set in a realistic scenario for a "managed by user type" column: BLOCK AFTER INSERT. If you can't see the row, then you can't create a new row. We will leave the FILTER predicate on for now:

```
--Note that you can alter a security policy, but it seems easier
--to drop and recreate in most cases.
DROP SECURITY POLICY rowLevelSecurity.Products_Product_SecurityPolicy;

CREATE SECURITY POLICY rowLevelSecurity.Products_Product_SecurityPolicy
    ADD FILTER PREDICATE rowLevelSecurity.Products_Product$SecurityPredicate(ProductType)
    ON Products.Product,
    ADD BLOCK PREDICATE rowLevelSecurity.Products_Product$SecurityPredicate(ProductType)
    ON Products.Product AFTER INSERT
    WITH (STATE = ON, SCHEMABINDING = ON);
```

Next we will test this by trying to create a new row of ProductType of 'snurf':

```
EXECUTE AS USER = 'valerie';

INSERT INTO Products.Product (ProductCode, Description, UnitPrice, ActualCost,ProductType)
VALUES  ('Test2' , 'Test2' , 100 , 100  , 'snurf');

REVERT;
```

```
Msg 33504, Level 16, State 1, Line 696
The attempted operation failed because the target object 'ClassicSecurityExample.Products.
Product' has a block predicate that conflicts with this operation. If the operation is
performed on a view, the block predicate might be enforced on the underlying table. Modify
the operation to target only the rows that are allowed by the block predicate.
```

One last configuration to demonstrate. In cases where the principal can see the row, we use the BEFORE type of predicate. In the following configuration, we will let the user see all rows, but only INSERT, UPDATE, or DELETE rows that meet the predicates:

```
DROP SECURITY POLICY rowLevelSecurity.Products_Product_SecurityPolicy;

CREATE SECURITY POLICY rowLevelSecurity.Products_Product_SecurityPolicy
    ADD BLOCK PREDICATE rowLevelSecurity.Products_Product$SecurityPredicate(ProductType)
      ON Products.Product AFTER INSERT,
    ADD BLOCK PREDICATE rowLevelSecurity.Products_Product$SecurityPredicate(ProductType)
      ON Products.Product BEFORE UPDATE,
```


WOW! eBook
www.wowebook.org

```
    ADD BLOCK PREDICATE rowLevelSecurity.Products_Product$SecurityPredicate(ProductType)
      ON Products.Product BEFORE DELETE
    WITH (STATE = ON, SCHEMABINDING = ON);
```

First, let's see what the data looks like to us now:

```
EXECUTE AS USER = 'valerie';
GO

SELECT *
FROM   Products.Product;
```

ProductId	ProductCode	Description	UnitPrice	ActualCost	ProductType
1	widget12	widget number 12	10.5000	8.5000	widget
2	snurf98	Snurfulator	99.9900	2.5000	snurf
5	Test	Test	100.0000	100.0000	snurf

Without changing back to the previous security context, try to DELETE the last row created:

```
DELETE Products.Product
WHERE    ProductCode = 'Test';
```

This fails:

```
Msg 33504, Level 16, State 1, Line 731
The attempted operation failed because the target object 'ClassicSecurityExample.Products.
Product' has a block predicate that conflicts with this operation. If the operation is
performed on a view, the block predicate might be enforced on the underlying table. Modify
the operation to target only the rows that are allowed by the block predicate.
```

Now, since we used BEFORE UPDATE in our predicate, we can update a row to a value that we may not be able to see after the update:

```
UPDATE Products.Product
SET    ProductType = 'snurf'
WHERE  ProductType = 'widget';

--We cannot update the row back, even though we can see it:

UPDATE Products.Product
SET    ProductType = 'widget'
WHERE  ProductType = 'snurf';

REVERT;
```


WOW! eBook
www.wowebook.org

This returns

```
Msg 33504, Level 16, State 1, Line 731
The attempted operation failed because the target object 'ClassicSecurityExample.Products.
Product' has a block predicate that conflicts with this operation. If the operation is
performed on a view, the block predicate might be enforced on the underlying table. Modify
the operation to target only the rows that are allowed by the block predicate.
```

Obviously we have just scratched the surface with the Row-Level Security feature, but you should be able to see that from here you could create a very tight layer of row-level security if needed.

■ **Tip** If you are using a single user to connect to the server, you can use the new `sys.sp_set_session_context` system procedure to save a security context to the connection metadata and the `SESSION_CONTEXT` system function to retrieve it. For example:

```
EXEC sys.sp_set_session_context @key = N'SecurityGroup', @value = 'Management';
```

and then you can use

```
SELECT SESSION_CONTEXT(N'SecurityGroup');
```

in your predicate function instead of `IS_MEMBER`, for example. Be sure that the actual user doesn't have ad hoc access to execute `sys.sp_set_session_context` or they can set whatever they want and circumvent security.

Using Data-Driven Row-Level Security

In this section, I want to expand on the predicate possibilities for both the view driven and row-level security feature driven row-level security implementation. Instead of embedding access through the code, we create a table that maps database role principals with some data in the table. In the case of our example scenarios, different types of products can be expanded and configured over time without any code needing to be released, giving total control to the process.

To start the process, we will create the following table that will hold the set of product types that a database role can see:

```
CREATE TABLE Products.ProductSecurity
(
    ProductType varchar(20), --at this point you probably will create a
                             --ProductType domain table, but this keeps the
                             --example a bit simpler
    DatabaseRole    sysname,
    CONSTRAINT PKProductsSecurity PRIMARY KEY(ProductType, DatabaseRole)
);
```


WOW! eBook
www.wowebook.org

Then, we 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 create a view named `ProductsSelective` view to show only rows to which the user has rights, based on row security:

```
ALTER VIEW Products.ProductSelective
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 exact same thing is available to the Row-Level Security feature as well. The predicate function can access tables as well. So you could create the function as follows (after dropping the policy that is using it, or you will get an error stating that you can't alter this function):

```
ALTER FUNCTION RowLevelSecurity.Products_Product$SecurityPredicate
                                                (@ProductType AS varchar(20))
    RETURNS TABLE
WITH SCHEMABINDING --not required, but a good idea nevertheless
AS
    RETURN (SELECT 1 AS Products_Product$SecurityPredicate
            WHERE is_member('db_owner') = 1
               OR  EXISTS (SELECT 1
                           FROM   Products.ProductSecurity
                           WHERE  ProductType = @ProductType
                             AND  IS_MEMBER(DatabaseRole) = 1));
```

The primary limitations are performance. Even with this being a new feature, I have heard of people using this feature with lots of rows in their row-level security tables, even using hierarchical data. While the sky is the limit, it is highly recommended before endeavoring to make heavy use of any row-level technique that you test heavily and with multiple users.

■ **Note** In the downloadable code, I fully implement these objects and include samples.

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 it does not always give you a great amount of fine-grained control. For example, if you want to let a user join to a table to get a value but not to browse the entire table using a `SELECT` statement, this would be very difficult using table/object-level security alone. However, by using T-SQL coded objects in a very similar manner as we have already done using views for row-level security, it is very possible.

WOW! eBook
www.wowebook.org

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, as seen earlier, horizontally by providing row-level security.

I put stored procedures together with views and functions in this 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 (though the row-level security policies will still be enforced). 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 they're 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 ad hoc or "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.

In security terms only, this allows you to have *situational control* over 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, I will limit the discussion to the security aspects.)

Take, for example, the case where a form 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 application window 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, let's 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
(
    GeneralValue varchar(20),
    SecretValue varchar(20)
);
GO
INSERT INTO ProcTest.Misc (GeneralValue, SecretValue)
```


WOW! eBook
www.wowebook.org

```
VALUES ('somevalue','secret'),
       ('anothervalue','secret');
```

Next, we will create a stored procedure to return the values from the `GeneralValue` column in the table, not the `SecretValue` column, and then grant rights to the `procUser` to execute the procedure:

```
CREATE PROCEDURE ProcTest.Misc$Select
AS
    SELECT GeneralValue
    FROM   ProcTest.Misc;
GO
GRANT EXECUTE on ProcTest.Misc$Select to ProcUser;
```

After that, we change the context to the `procUser` user and try to `SELECT` from the table:

```
EXECUTE AS USER = 'ProcUser';
GO
SELECT GeneralValue , SecretValue
FROM   ProcTest.Misc;
```

We get the following error message, because the user hasn't been given rights to access this table:

```
Msg 229, Level 14, State 5, Line 768
The SELECT permission was denied on the object 'Misc', database 'ClassicSecurityExample',
schema 'ProcTest'.
```

However, when we execute the following procedure:

```
EXECUTE ProcTest.Misc$Select;
```

we get the expected result, that the user does have access to execute the procedure:

```
GeneralValue
--------------------
somevalue
anothervalue
```

This is one of the best ways to architect a database solution, both for security and for performance. 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. (For performance it allows caching of complex plans and having a known set of queries to tune, but more on that in Chapter 13.)

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;
```


WOW! eBook
www.wowebook.org

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 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 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 an object 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 clause to the object, you can change the security context of a procedure to a different server or database principal when the execution begins:

```
CREATE <objectType> <schemaName>.<objectName>
WITH EXECUTE AS <'userName' | caller | self | owner>; --adding this
```

The different options for whom to execute as are as follows:

- 'userName': A specific 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.
- self: It's in the context of the user who created the procedure. You can see who self will represent by looking in sys.sql_modules at the execute_as_principal_id column.
- 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 EXECUTE AS setting come into play. The following statements go along with the EXECUTE AS clause:

- EXECUTE AS CALLER: If you are using a security context other than 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.

WOW! eBook
www.wowebook.org

As an example, I'll show you how to build a scenario where one schema owner contains a table and another schema owner has a table and a procedure that the schema owner wants to use to access the first user's table. Finally, the scenario has an average user who wants to execute the stored procedure.

■ **Caution** This next example is not intended as a "best practice" for most databases, but rather as an example of how ownership chaining works with objects. Ideally, all objects are owned by the database owner for typical databases. Of course, not all databases are typical, so your usage may differ. Also note that I switch context of user several times to get different ownership chaining to occur. It can be tricky, which is why the first paragraph of this caution is what it is!

First, 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 schema where one of the objects resides.
- `ProcedureOwner`: This user is owner of a table 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, 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 NOT NULL CONSTRAINT PKPerson PRIMARY KEY,
    FirstName   varchar(20) NOT NULL,
    LastName    varchar(20) NOT NULL
);
```


WOW! eBook
www.wowebook.org

```
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, 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 without giving ShemaOwner impersonation rights. Here I
       --step back to the db_owner user you have used throughout the chapter
GO
EXECUTE AS USER = 'ProcedureOwner';
```

Then, 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 NOT NULL CONSTRAINT PKOtherPerson PRIMARY KEY,
    FirstName   varchar(20) NOT NULL,
    LastName    varchar(20) NOT NULL
);
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');
```


WOW! eBook
www.wowebook.org

This returns the following:

```
table              schema                 owner
------------------ ---------------------- ------------------------------
OtherPerson        ProcedureOwnerSchema   ProcedureOwner
Person             SchemaOwnersSchema     SchemaOwner
```

Next, create two procedures as the procedureOwner user, one for the WITH EXECUTE 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, grant rights on the procedure to the AveSchlub user:

```
GRANT EXECUTE ON ProcedureOwnerSchema.Person$asCaller TO AveSchlub;
GRANT EXECUTE ON ProcedureOwnerSchema.Person$asSelf TO AveSchlub;
```

Then, change to the context of the AveSchlub:

```
REVERT; EXECUTE AS USER = 'AveSchlub'; --If you receive error about not being able to
--impersonate another user, it means you are not executing as dbo..
```

Finally, execute the procedure:

```
--this proc is in context of the caller, in this case, AveSchlub
EXECUTE ProcedureOwnerSchema.Person$asCaller;
```


WOW! eBook
www.wowebook.org

This produces the following output, because the ownership chain is fine for the ProcedureOwnerSchema object, but not for the SchemaOwnersSchema:

```
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'.
```

Next, 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 ProcedureOwner, not the context of the caller, aveSchlub. Using EXECUTE AS to change security context is a cool, powerful 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 during 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 without having to worry about ownership chaining (I will discuss this more in Chapter 13 when I discuss code-level design, but generally these are calls that are formed as textual queries in your stored procedures, rather than being compiled), 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
```


WOW! eBook
www.wowebook.org

```
 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 (or rights to create a table), 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;
```

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 db_owner's role:

```
Msg 262, Level 14, State 1, Procedure testDboRights, Line 5
CREATE TABLE permission denied in database 'ClassicSecurityExample'.
```

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.

WOW! eBook
www.wowebook.org

■ **Tip** As will be discussed in the "Crossing Database Lines" section later in this chapter, to use external resources (like a table in a different database) using impersonation of a database principal, you need to set `TRUSTWORTHY` to `ON` using the `ALTER DATABASE` command.

In 2016, the Row-Level Security feature throws a bit of a spanner into how this all works. Usually, when you execute a coded object (or use a view), you get access to everything the owner of the object provided you. However, for Row-Level Security, when executing as the caller, the caller's information is provided to the predicate function. When executing as a different user, that user's information is used.

As an example, create the following table, with a few rows:

```
CREATE TABLE dbo.TestRowLevelChaining
(
        Value    int CONSTRAINT PKTestRowLevelChaining PRIMARY KEY
)
INSERT dbo.TestRowLevelChaining (Value)
VALUES  (1),(2),(3),(4),(5);
```

Then set up a filtering predicate that returns values > 3 if you are not the dbo, and apply it:

```
CREATE FUNCTION RowLevelSecurity.dbo_TestRowLevelChaining$SecurityPredicate
                                        (@Value AS int)
RETURNS TABLE WITH SCHEMABINDING
AS RETURN (SELECT 1 AS dbo_TestRowLevelChaining$SecurityPredicate
           WHERE  @Value > 3 OR  USER_NAME() = 'dbo');
GO
CREATE SECURITY POLICY RowLevelSecurity.dbo_TestRowLevelChaining_SecurityPolicy
    ADD FILTER PREDICATE RowLevelSecurity.dbo_TestRowLevelChaining$SecurityPredicate (Value)
    ON dbo.TestRowLevelChaining WITH (STATE = ON, SCHEMABINDING = ON);
```

Then set up two procedures, one that executes as caller, and the other that executes as the dbo user:

```
CREATE PROCEDURE dbo.TestRowLevelChaining_asCaller
AS
SELECT * FROM dbo.TestRowLevelChaining;
GO

CREATE PROCEDURE dbo.TestRowLevelChaining_asDbo
WITH EXECUTE AS  'dbo'
AS
SELECT * FROM dbo.TestRowLevelChaining;
```

Create a new user, and grant it rights to execute both procedures:

```
CREATE USER Bobby WITHOUT LOGIN;
GRANT EXECUTE ON dbo.TestRowLevelChaining_asCaller TO Bobby;
GRANT EXECUTE ON dbo.TestRowLevelChaining_asDbo TO Bobby;
```


WOW! eBook
www.wowebook.org

Now, executing the new user, you will see that the `asCaller` variant only returns the two rows that are greater than 3:

```
EXECUTE AS USER = 'Bobby'
GO
EXECUTE  dbo.TestRowLevelChaining_asCaller;
```

```
Value
-----------
4
5
```

But the `asDbo` variant is now executing as if the `dbo` user is the one executing it, even for the row-level security access:

```
EXECUTE  dbo.TestRowLevelChaining_asDbo;
```

This returns all of the data in the table:

```
Value
-----------
1
2
3
4
5
```

This is a very useful thing, first that the normal case allows you to largely ignore row-level security when writing your objects, and second that you can apply it to objects that already exist. It is essential to understand how row-level security applies to cases when you are using impersonation, so as to not expect to be giving access to an object and accidentally grant rights to a lot more data than you are expecting.

Views and Table-Valued Functions

Views and table-valued functions share security aspects with executable coded objects, but behave in most contexts like tables. 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. You can use views to provide an encapsulation layer that looks like tables to the user (and using triggers like we discussed in Chapter 7, can almost always be programmed to behave like actual tables with enough work). Table-valued functions are very similar to views, but the data they return cannot be modified even when they are of the simple, single-statement variety.

In this section, I'll briefly discuss a few ways for views and table-valued functions to encapsulate data in a manner that leaves 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. 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.

WOW! eBook
www.wowebook.org

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 11, I will give some basic coverage of application architecture.)

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.

We'll use two properties of views to build a more secure database. The first is to assign 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 (if you have additional rows from testing, delete them now; a statement to do so is provided in the download):

```
ProductId   ProductCode Description          UnitPrice    ActualCost    ProductType
----------- ----------- -------------------- ------------ ------------ ------------------1
            widget12    widget number 12     10.5000      8.5000        widget
2           snurf98     Snurfulator          99.9900      2.5000        snurf
```

We could construct a view on this:

```
CREATE VIEW Products.AllProducts
AS
SELECT ProductId,ProductCode, Description, UnitPrice, ActualCost, ProductType
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 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. As the view is made up of one table, it is updateable as well.

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. (Earlier we used horizontal partitioning for 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;
```


WOW! eBook
www.wowebook.org

In the same manner, 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
```

Now for each of these views and functions, you can simply `GRANT SELECT` rights to a user to use them, and almost all tools would be able to use them just like tables.

Crossing Database Lines

So far, most of the code and issues we've discussed have been concerned with everything owned by a single owner and everything has been within a single database. This will almost always be the desired pattern of development, but sometimes it is not possible to achieve. 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 SQL DB, contained databases, and even the developer tools that are shipped with SQL Server. (Azure SQL DB does allow some cross-database queries with its elastic database query features, which you can read about here: azure.microsoft.com/en-us/blog/querying-remote-databases-in-azure-sql-db/.) Sometimes, however, 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.

WOW! eBook
www.wowebook.org

A typical example of cross-database access 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. A second scenario could be a hosted server with databases from many clients, but clients need access to more than one database without gaining access to any other client's data.

Beyond the coding and maintenance aspects of cross-database access, which aren't necessarily trivial, the most complex 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. `UserA` in `database1` is not exactly 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 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 owned by the same login. So, not only does the object creator need to have access to the objects outside the database, the caller needs rights also.

This section demonstrates four different concepts for the on-premises version of SQL Server when dealing with accessing data outside of a single database.:

- Using cross-database chaining
- Using impersonation to implement cross-database connections
- Using a certificate-based trust
- 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 when you do, you will want to choose the method that is the most secure for your needs.

Using Cross-Database Chaining

The cross-database chaining solution is to tell the database to recognize that if the owners of `database1` and `database2` are the same, it should not let ownership chaining be broken when crossing database boundaries. Then, if you, as system administrator, want to allow users to use your objects seamlessly across databases, that'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 an internal server. For example, we have several databases that make up a complete 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, as will be described in more detail in Chapter 14).

WOW! eBook
www.wowebook.org

■ **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.

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, the following scenario creates two databases with a table in each database and then a procedure. First, create the new database and add a simple table. You don't need to 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. You will start with this database in an uncontained model and then 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 PapaSmurf WITH PASSWORD = 'La la, la la la la, la, la la la la';
CREATE USER  PapaSmurf FROM LOGIN PapaSmurf;
CREATE TABLE dbo.Table1 ( Value int );
```

Next, 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 PapaSmurf FROM LOGIN PapaSmurf;
```

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. If you are using these techniques on a shared server, using `sa` would not be the preferred method, and you could use different database owners for different users. Do this with the `ALTER AUTHORIZATION DDL` statement:

WOW! eBook
www.wowebook.org

```
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, as it is essential that the databases are owned by the same server principal for the upcoming examples:

```
name              owner
----------------- ------------------
ExternalDb        sa
LocalDb           sa
```

Next, 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 your new user:

```
CREATE PROCEDURE dbo.ExternalDb$TestCrossDatabase
AS
SELECT Value
FROM   ExternalDb.dbo.Table1;
GO
GRANT EXECUTE ON dbo.ExternalDb$TestCrossDatabase TO PapaSmurf;
```

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 `PapaSmurf` that is in the `localDb`:

```
EXECUTE AS USER = 'PapaSmurf';
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 "PapaSmurf" is not able to access the database "ExternalDb" under the
current security context.
```


WOW! eBook
www.wowebook.org

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 --It does not need to be trustworthy since it is not reaching out
   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 you set up with the ALTER AUTHORIZATION statements.
- The user has access to connect to the external database, which is why you created the user when you set up the externalDb database. (You can also use the guest user 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 by 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
```

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 using this method, 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 check 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 = 'PapaSmurf';
go
EXECUTE dbo.ExternalDb$TestCrossDatabase;
```


WOW! eBook
www.wowebook.org

```
GO
REVERT;
GO
```

You will see that it behaves the exact same way and gives you a result to the query. However, connecting with a contained user is a different challenge. First, create a contained user, and then give it rights to execute the procedure:

```
CREATE USER Gargamel WITH PASSWORD = 'Nasty1$';
GO
GRANT EXECUTE ON dbo.ExternalDb$TestCrossDatabase to Gargamel;
```

Next, change to the database security context of the new contained user and try to change context to the externalDb:

```
EXECUTE AS USER = 'Gargamel';
GO
USE ExternalDb;
```

This will give you the following error (plus or minus some characters in that server principal moniker, naturally):

```
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 dynamic management view. To find objects that reference outside data, you can use the following query:

```
SELECT  OBJECT_NAME(major_id) AS object_name,statement_line_number,
        statement_type, feature_name, feature_type_name
FROM    sys.dm_db_uncontained_entities
WHERE   class_desc = 'OBJECT_OR_COLUMN';
```


WOW! eBook
www.wowebook.org

For our database, it will return the following, which corresponds to the procedure 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
WHERE   class_desc = 'DATABASE_PRINCIPAL'
  and   USER_NAME(major_id) <> 'dbo';
```

And you created one already in this chapter:

```
  USER_NAME
  --------------------
  PapaSmurf
```

■ **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, to do a bit of housekeeping and remove containment from the database, delete the contained user you created and turn off containment (you have to drop the user or, as previously mentioned, you would receive an error stating that uncontained databases cannot have contained users):

```
DROP USER Gargamel;
GO
USE Master;
GO
ALTER DATABASE localDB  SET CONTAINMENT = NONE;
GO
USE LocalDb;
GO
```


WOW! eBook
www.wowebook.org

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, since you will be executing code that reaches out of the current database.

```
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 PapaSmurf;
```

If the login of the owner of the dbo schema (in this example 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.

Because setting TRUSTWORTHY to ON requires sysadmin privileges, using impersonation isn't a tremendous hole, but note that the members of the sysadmin role aren't required to understand the implications if one of their users calls up and asks for TRUSTWORTHY to be turned on for them.

Now, when you execute the procedure as PapaSmurf user, it works:

```
EXECUTE AS USER = 'PapaSmurf';
GO
EXECUTE dbo.ExternalDb$TestCrossDatabase_Impersonation;
GO
REVERT;
```

If you toggle TRUSTWORTHY to OFF and try to execute the procedure

```
ALTER DATABASE localDb  SET TRUSTWORTHY OFF;
GO
EXECUTE dbo.ExternalDb$TestCrossDatabase_Impersonation;
```


WOW! eBook
www.wowebook.org

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.
```

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, go back to the containment method. Turn back on TRUSTWORTHY, set the containment, and re-create the Gargamel user, giving rights to the impersonation procedure:

```
ALTER DATABASE LocalDb  SET TRUSTWORTHY ON;
GO
ALTER DATABASE LocalDB  SET CONTAINMENT = PARTIAL;
GO
CREATE USER Gargamel WITH PASSWORD = 'Nasty1$';
GO
GRANT EXECUTE ON ExternalDb$testCrossDatabase_Impersonation TO Gargamel;
```

Now execute the procedure in the context of the contained user:

```
EXECUTE AS USER = 'Gargamel';
GO
EXECUTE dbo.ExternalDb$TestCrossDatabase_Impersonation;
GO
REVERT;
```

This time, you will see that 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, clean up the users and containment as you have done before:

```
DROP USER Gargamel;
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. You'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

WOW! eBook
www.wowebook.org

rights in the externalDB. Because the certificate matches, SQL Server knows that this cross-database access is acceptable.

First, turn off the TRUSTWORTHY setting:

```
USE LocalDb;
GO
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, create another procedure and give the user PapaSmurf 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 PapaSmurf;
```

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';
```


WOW! eBook
www.wowebook.org

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;
```

Now, you're good to go. Change back to the LocalDb and execute the procedure:

```
USE LocalDb;
GO
EXECUTE AS LOGIN = 'PapaSmurf';
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 far more secure, for certain. Pretty much the major downside to this is that it does not work with views. 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."

This method does not work when using containment, unless you use impersonation, since the user will not have any way to access the other database. (Code included in download.)

Finally, clean up the databases used for the examples and move back to the ClassicSecurityExample database you 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 OLE DB is being deprecated and will not be supported forever. For more details,

WOW! eBook
www.wowebook.org

check 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 from the executing user when crossing SQL Server instance connections and certainly when using any data source that isn't SQL Server–based. You will configure the user that is accessing the external data in the configuration. Using 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, nor even desirable, 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 well-designed security granularity, a few users will still need to have rights to run as a member of the `sys_admin` server role, giving them access to *all* data.

On the more administrative side of things, if DBAs have access to unencrypted backups of a database, they may be able to easily access any data in a database by simply restoring it to a different server where they are administrators. 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?

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). In this section we will look at two methods of obfuscating data:

- *Encryption*: Encrypting the data so no one without the certificates, passwords, etc. can reasonably access the data. (Encryption isn't perfect, given enough time, computing power, and knowhow to crack it.)
- *Dynamic data masking*: A new feature in 2016 that allows you to mask over data in given scenarios.

WOW! eBook
www.wowebook.org

Neither of these techniques is perfect, but together you can provide a very solid obfuscation layer over your data.

Encrypting Data

The following are several tools in the SQL Server arsenal to provide encryption for your data. I will not cover any of them in great detail, but if you are architecting a database solution, it is fairly essential that you know about these technologies.

- *Transparent Data Encryption*: A feature that 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).
- *Encrypted backups*: SQL Server provides methods using the `BACKUP` command to make backups encrypted.
- *Encryption functions*: Functions that let you encrypt values using SQL Server function calls. They are very simple to use, and the one of these features I will demonstrate. The biggest downfall is that all of the data needed to decrypt the data is stored locally to SQL Server.
- *Always Encrypted*: A new feature in SQL Server 2016 that is somewhat external to the database engine. Unlike the encryption functions, the information needed to decrypt the data is not stored in SQL Server.
- *Binary columns*: Not technically a feature of encryption, but prior to Always Encrypted, one solution was simply to encrypt the data externally to SQL Server and store the binary value in a column. Old school is still sometimes the best school.

The biggest "key" to any encryption strategy that is intended to obfuscate important data is that you don't store all the information to decrypt data together with the data. It would be just as silly as taking the time to install deadbolt locks and then leaving a key on the outside of what you are protecting on a little cup hook.

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. You just need to consider two things when pondering your encryption needs, First: "Who can decrypt the data, and how?" If the information to do the decryption is stored with the encrypted data, you are not gaining much. SQL Server's basic encryption uses 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 (EKM), 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. Second: If someone gets the data, how easily could they decrypt it with unlimited computing power and time? Some encryption is deterministic, in that every time value 'A' is encrypted, the same binary string is output. This can make things like searching for encrypted values faster, but makes it easier to break. Probabilistic encryption schemes output different values every time, but will be really difficult to search for an encrypted value.

A very useful and security-minded goal you should have in most organizations is to try to avoid holding sensitive information (like credit card numbers) in your database at all and use a third-party service to handle it for you. Systems like this will give you a token back that only one organization can make use of. 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 data is stolen, the goal needs to be that the data is worthless, so that all your press release needs to 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?

WOW! eBook
www.wowebook.org

Using Dynamic Data Masking to Hide Data from Users

Dynamic data masking is used to obfuscate data from users, but will do nothing for the security of your data if it is stolen. What it does is, allow you to show a user a column, but instead of showing them the actual data, it masks it from their view. As an example, consider a table that has e-mail addresses. You might want to mask the data so most users can't see the actual data when they are querying the data. Dynamic data masking falls under the head of Security Center features in Books Online (https://msdn.microsoft.com/en-us/library/mt130841.aspx), but as we will see, it doesn't behave like classic security features, as you will be adding some code to the DDL of the table.

There are a couple of limitations. First, masks only pertain to SELECT operations. If a user can INSERT, UPDATE, or DELETE the row, they still can even if the data is masked. Hence, this feature is generally going to be used strictly for read operations. The second, fairly huge, limitation with this feature is that there is only one right to allow unmasked viewing of all masked data, and it is at the database level (not the column level, nor the table level).

To demonstrate, consider the following table (including my favorite column I have ever included in a demo, YachtCount!):

```
CREATE SCHEMA Demo;
GO
CREATE TABLE Demo.Person --warning, I am using very small column datatypes in this
                         --example to make looking at the output easier, not as proper sizes
(
    PersonId    int NOT NULL CONSTRAINT PKPerson PRIMARY KEY,
    FirstName    nvarchar(10) NULL,
    LastName    nvarchar(10) NULL,
    PersonNumber varchar(10) NOT NULL,
    StatusCode    varchar(10) CONSTRAINT DFLTPersonStatus DEFAULT ('New')
                          CONSTRAINT CHKPersonStatus
                                  CHECK (StatusCode in ('Active','Inactive','New')),
    EmailAddress nvarchar(40) NULL,
    InceptionTime date NOT NULL, --Time we first saw this person. Usually the row create
    time,
                                --but not always
    -- YachtCount is a number that I didn't feel could insult anyone of any     -- origin,
       ability, etc that I could put in this table
    YachtCount   tinyint NOT NULL CONSTRAINT DFLTPersonYachtCount DEFAULT (0)
                          CONSTRAINT CHKPersonYachtCount CHECK (YachtCount >= 0),
);
```

Some of this data we will want to keep hidden from viewers. The PersonNumber, StatusCode, EmailAddress, InceptionTime, and YachtCount values all need to be hidden away. There are four types of built-in masks that we can use:

- *Default*: Takes the default mask of the datatype (*not* the default of the column)
- *Email*: Masks the e-mail so you only see a few meaningful characters to give you an idea of what the e-mail address is, but not the full address
- *Random*: Puts a random number in place of an actual number (which can actually be kind of weird, as we will see)
- *Custom String (Partial)*: Basically gives you control over what characters to keep and what to replace them with

WOW! eBook
www.wowebook.org

Dynamic data masking becomes part of the table itself, so we will use `ALTER COLUMN` to add this to the columns mentioned, starting with using default for each of the columns:

```
ALTER TABLE Demo.Person ALTER COLUMN PersonNumber
    ADD MASKED WITH (Function = 'default()');
ALTER TABLE Demo.Person ALTER COLUMN StatusCode
    ADD MASKED WITH (Function = 'default()');
ALTER TABLE Demo.Person ALTER COLUMN EmailAddress
    ADD MASKED WITH (Function = 'default()');
ALTER TABLE Demo.Person ALTER COLUMN InceptionTime
    ADD MASKED WITH (Function = 'default()');
ALTER TABLE Demo.Person ALTER COLUMN YachtCount
    ADD MASKED WITH (Function = 'default()');
```

Next, we add a few rows:

```
INSERT INTO Demo.Person (PersonId,FirstName,LastName,PersonNumber, StatusCode,
                         EmailAddress, InceptionTime,YachtCount)
VALUES(1,'Fred','Washington','0000000014','Active','fred.washington@ttt.net','1/1/1959',0),
      (2,'Barney','Lincoln','0000000032','Active','barneylincoln@aol.com','8/1/1960',1),
      (3,'Wilma','Reagan','0000000102','Active',NULL, '1/1/1959', 1);
```

Next, we create a user that will be masked, and give the user `SELECT` rights to the table:

```
CREATE USER MaskedMarauder WITHOUT LOGIN;
GRANT SELECT ON Demo.Person TO MaskedMarauder;
```

Then, we select the data as the `sysadmin` user and select it again impersonating the `MaskedMarauder`:

```
SELECT PersonId, PersonNumber, StatusCode, EmailAddress, InceptionTime, YachtCount
FROM   Demo.Person;

EXECUTE AS USER = 'MaskedMarauder';

SELECT PersonId, PersonNumber, StatusCode, EmailAddress, InceptionTime, YachtCount
FROM   Demo.Person;

REVERT;
```

This returns two result sets, the first unmasked, the second masked:

```
PersonId    PersonNumber StatusCode EmailAddress                InceptionTime YachtCount
----------- ------------ ---------- ---------------------------- ------------- ----------
1           0000000014   Active     fred.washington@ttt.net      1959-01-01    0
2           0000000032   Active     barneylincoln@aol.com        1960-08-01    1
3           0000000102   Active     NULL                         1959-01-01    1

PersonId    PersonNumber StatusCode EmailAddress                InceptionTime YachtCount
----------- ------------ ---------- ---------------------------- ------------- ----------
1           xxxx         xxxx       xxxx                         1900-01-01    0
2           xxxx         xxxx       xxxx                         1900-01-01    0
3           xxxx         xxxx       NULL                         1900-01-01    0
```


WOW! eBook
www.wowebook.org

A few initial notes. First, the data looks like data in some cases. `YachtCount = 0` is the data for at least one row. Second, `NULL` data is still shown as `NULL`. Some defaults work nicely for the general case, like string being `'xxxx'`. However, we probably want a bit more information in some cases (`emailAddress`, `personNumber`), and we will use a random value for `YachtCount`.

First we will change the `emailAddress` masking. Instead of default, we will use `email()` as the masking function:

```
ALTER TABLE Demo.Person ALTER COLUMN EmailAddress
    ADD MASKED WITH (Function = 'email()');
```

`SELECT`ing the data as the `MaskedMarauder` (a name which is not getting old to type), we see that email address gives us enough information to perhaps verify with a customer (Note that all email addresses end in .com):

```
PersonId    PersonNumber StatusCode EmailAddress                 InceptionTime YachtCount
----------- ------------ ---------- ------------------------------ ------------ ----------
1           xxxx         xxxx       fXXX@XXXX.com                 1900-01-01    0
2           xxxx         xxxx       bXXX@XXXX.com                 1900-01-01    0
3           xxxx         xxxx       NULL                          1900-01-01    0
```

But we can do better. Next we will try the random masking function, with `YachtCount`. Random only works with numeric data:

```
ALTER TABLE Demo.Person ALTER COLUMN YachtCount
    ADD MASKED WITH (Function = 'random(1,100)'); --make the value between 1 and 100.
```

Viewing the masked versions of the data now shows the following nonsensical, definitely noncontroversial, data:

```
PersonId    PersonNumber StatusCode EmailAddress                 InceptionTime
YachtCount
----------- ------------ ---------- ------------------------------ ------------ ----------
1           xxxx         xxxx       fXXX@XXXX.com                 1900-01-01    45
2           xxxx         xxxx       bXXX@XXXX.com                 1900-01-01    74
3           xxxx         xxxx       NULL                          1900-01-01    42
```

Finally, we will use the custom string masking function, `partial()`. This lets you mask all or some of the data in a string based columns using a function: partial (number of characters to keep at the start of the string, replace with string, number of characters to keep on the end of the string). So we are going to mask the person number to show the first and last characters, and then all status codes with `'Unknown'`:

```
ALTER TABLE Demo.Person ALTER COLUMN PersonNumber
    ADD MASKED WITH (Function = 'partial(1,"-------",2)'); --note double quotes on the text

ALTER TABLE Demo.Person ALTER COLUMN StatusCode
    ADD MASKED WITH (Function = 'partial(0,"Unknown",0)');
```


WOW! eBook
www.wowebook.org

Here is our final output, though the YachtCount may change:

```
PersonId    PersonNumber StatusCode EmailAddress                InceptionTime YachtCount
----------- ------------ ---------- --------------------------- ------------- ----------
1           0-------14   Unknown    fXXX@XXXX.com               1900-01-01    26
2           0-------32   Unknown    bXXX@XXXX.com               1900-01-01    9
3           0-------02   Unknown    NULL                        1900-01-01    15
```

As mentioned earlier in the section, if we want to allow a user to see the unmasked data, there are two ways. First, we can grant the UNMASK permission at the database level. This means the user will be able to see all data in the database unmasked. The second is basically using impersonation. Create a user that has UNMASK rights, and let the user see the data that way.

While it is limited in general-purpose value, dynamic data masking may come in handy in some situations where you want to hide the actual values of columns in tables most of the time, particularly when you want to show the user partial values.

Auditing SQL Server Use

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?"

Ideally though, auditing is not done as an alternative to implementing a full-blown security system, but simply to watch what users do, in case they do something they shouldn't. To implement our Big Brother security scenario, I'll configure a server and database audit using SQL Server's audit features. There are other methods, like adding triggers (DML and DDL), or using Extended Events. I won't cover them here, but they can be alternatives to using the audit feature if you cannot use audit for some reason (like database audit not being in Standard edition, or if you are using Azure SQL DB).

■ **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 a 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.

SQL Server Audit is a tremendously cool feature that will allow you to implement detailed monitoring in a declarative manner 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. Using SQL Server Audit, you will be able to watch what the user is doing. If you want to capture what is changing in the data, you can use the Temporal feature (SQL Server 2016), or a DML trigger, as we discussed in Chapter 8.

WOW! eBook
www.wowebook.org

■ **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 and right-click the script option.

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 queued for audit or not executed. It doesn't write directly to the file; 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% up to date).

The audit structures consist of three basic layers of 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, we 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), watch for the `employee` or `manager` user to execute a `SELECT` statement against the `Products.Product` table, and watch for `SELECT`s by anyone on `Sales.Invoice`. First, we define the `SERVER AUDIT`:

```
USE master;
GO
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 --of each file
      ,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.

WOW! eBook
www.wowebook.org

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.

At the server level, you can watch for changes to the configuration of your server. In the example, we are going to watch for server principals to change:

```
ALTER SERVER AUDIT SPECIFICATION ProSQLServerDatabaseDesign_Server_Audit
    ADD (SERVER_PRINCIPAL_CHANGE_GROUP);
```

Next, we will go through the same process for the database that we did for the server, setting up the container for the audit using the DATABASE AUDIT SPECIFICATION command for the table we created earlier. At the database level, we can look for configuration changes, but perhaps more interestingly, we can audit people accessing tables. In this example, we will monitor the one thing that is pretty difficult to do with any other method, auditing certain users SELECTing data from a table:

```
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 users use of the Products.Product table and the Products.AllProducts view (created earlier in this chapter). 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);
```

Enabling an Audit Specification

Finally, we enable the two audit specifications that we've just created. Note that, 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);
```


WOW! eBook
www.wowebook.org

```
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 of 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 = 'A very gOOd password!';
GO
USE ClassicSecurityExample;
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);
```


WOW! eBook
www.wowebook.org

we can see the different statements that were executed (and see the two statements where the permission failed):

```
event_time              succeeded database_principal_name  statement
----------------------- --------- ------------------------ ---------------------------------
2016-09-02 03:36:53.31 1
2016-09-02 03:36:53.37 1         dbo                      CREATE LOGIN MrSmith WITH PASS...
2016-09-02 03:36:53.40 1         Manager                  SELECT *
                                                          FROM   Products.Product
2016-09-02 03:36:53.58 0         Manager                  SELECT  *
                                                          FROM    Products.AllProducts  ...
2016-09-02 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 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 (table/view/etc.) 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, the following is returned:

```
audit_specification_name                        audit_action_name
----------------------------------------------- ----------------------------------------
ProSQLServerDatabaseDesign_Server_Audit         SERVER_PRINCIPAL_CHANGE_GROUP
```


WOW! eBook
www.wowebook.org

Digging deeper, to get the objects and actions, the following query will get you the database-level actions that are being audited:

```
SELECT audit_action_name,dp.name AS [principal],
       SCHEMA_NAME(o.schema_id) + '.' + o.name AS object
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 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 new feature that is useful for DBAs/architects who have the need to audit the activities of their users and especially administrators.

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 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 user a member of the same role, rather than adding rights directly to the user. Ideally, the roles in your database are the same in development, test, and production, but the users who are members of those roles will likely change from environment to environment.
- *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.

WOW! eBook
www.wowebook.org

- *Consider security using stored procedures as the access layer*: 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. Chapter 13 will advance this concept further.
- *Don't overuse the impersonation features*: `EXECUTE AS` is a blessing, and it opens 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.
- *Segregate security between environments*: Security in development environments will be very different. Take care not to end up with developers having 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.

Summary

Security is a large topic, and understanding all the implications requires way more information 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: logins, users, roles, certificates, 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 hard to view the data unless you specifically try to, generally by using encryption to make the data unreadable without a key.

WOW! eBook
www.wowebook.org

- Next, we discussed using an audit trail to give 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).

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.

WOW! eBook
www.wowebook.org

CHAPTER 10

Index Structures and Application

In this life, we have to make many choices. Some are very important choices. Some are not. Many of our choices are between good and evil. The choices we make, however, determine to a large extent our happiness or our unhappiness, because we have to live with the consequences of our choices.

— James E. Faust,
American religious leader, lawyer, and politician

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 based on a column or two, do a little math perhaps, and give me back information (and naturally do it incredibly fast, if you don't mind). While simple isn't always the case, the engine usually obliges with an answer extremely quickly. Usually, but not always. This is where the DBA and data programmers must figure out what the optimizer is doing and help it along. If you are not a technical person, you probably think of query tuning as magic. It is not. What it is 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 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 in this chapter to show you how your design choices can affect the way work gets done.

Our part in the process of getting back lightning-fast answers to complex questions 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). We do this first by designing and implementing as close to the relational model as possible by normalizing our structures, using good set-based code (no cursors), following best practices with coding T-SQL, and so on. This is obviously 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, Third Edition,* 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 we have built our systems correctly, the next step is to help out by adjusting the physical structures using proper hardware laid out well on disk subsystems, filegroups, files, partitioning, and configuring our hardware to work best with SQL Server, and with the load we will be, are, and have been placing on the server. This is yet another book's worth of information, for which I will direct you to a book I recently reviewed as a technical editor, Peter Carter's *Pro SQL Server Administration* (Apress, 2015).

In this chapter we are going to focus primarily on the structures that you will be adding and subtracting regularly: indexes. Indexing is a constant balance of helping performance versus harming performance. If you don't use indexes enough, queries 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

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_10

WOW! eBook
www.wowebook.org

we will cover in Chapter 11 by forcing the query processor to lock a lot more memory 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 server needs to refill your glass immediately, depending on the situation.)

All plans that I present will be obtained using the `SET SHOWPLAN_TEXT ON` statement. When you're doing this locally, it is almost always 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`) commands does 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.

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 almost all of what you need to make normal OLTP queries run nicely, since the most common searches that people will do will be on identifying information.

What complicates all of our choices at this point in the process in the fifth edition of this book is that Microsoft has added an additional engine to the mix, in both the box and cloud versions of the product. I have noted in Chapters 6 and onward that these different versions exist, and that some of the sample code from those chapters will have downloaded versions to show how it might be coded with the different engines. But the differences are more than code. The design of the logical database isn't that much different, and what is put into the physical database needn't change considerably either. But there are internal differences to how indexes work that will change some of the implementation, which I will explore in this chapter. Chapter 11 will discuss how these changes affect concurrency and isolation, and finally, Chapter 13 will discuss changes in coding you will see. For now, we will start with the more common on-disk indexes, and then we will look at the memory-optimized index technologies, including indexes on in-memory OLTP table and columnstore indexes (which will be discussed in more detail in Chapter 14 as well).

Indexing 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 foretell 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.

WOW! eBook
www.wowebook.org

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. The point is that, situationally, you get far greater gain from writing better code than you do from just throwing hardware at the problem. The ideal situation is adequate hardware and excellent code, naturally.)

Basic Index Structure

An index is a structure that SQL Server can maintain to optimize access to the physical data in a table. An index can be 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, when you are looking for one specific row (and you know that the value you are searching for is unique), 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.

As an example, consider that you have a completely unordered list of employees and their details in a book. 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';
```


WOW! eBook
www.wowebook.org

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, 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. (And as we will notice in the next chapter, the more rows SQL Server needs to touch, the more blocking you may do to other users.)

If we instead created an index on the `LastName` column, the index creates a structure to allow searching for the rows with the matching `LastName` in a logical fashion 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, most indexes are implemented using what is known as a *balanced tree* (B-tree) structure (some others are built using *hash* structures, as we cover when we get to the in-memory OLTP section, but B-trees will be overwhelmingly the norm, so they get the introduction). The B-tree 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 pages on last level in the index are referred to as the *leaf pages*, which contain 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 millions of values in the index.

Figure 10-1 shows an example of the type of B-tree that SQL Server uses for on-disk indexes (memory-optimized indexes are different and will be covered later). 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 will fit physically on the page. 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 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.

WOW! eBook
www.wowebook.org

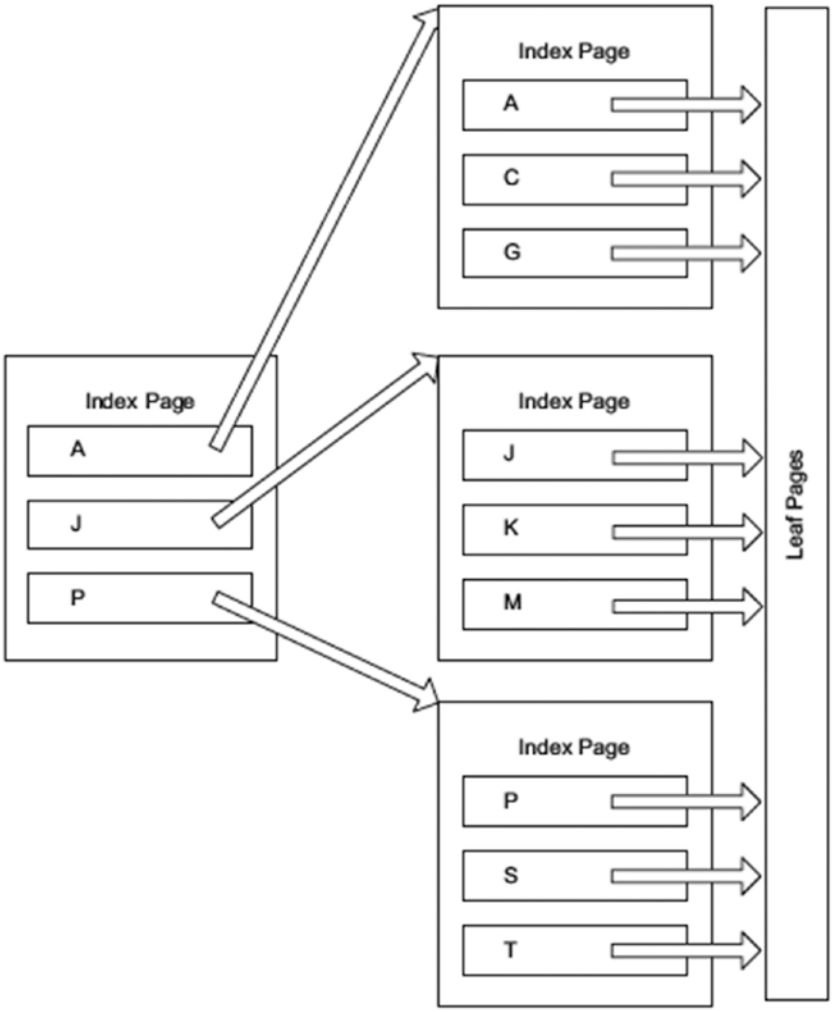


Figure 10-1. *Basic index structure*

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 1,700 bytes for some types of indexes), it's possible to have anywhere from 4 entries to over 1,000 on a single page. The more keys you can fit onto a single index page, allows that page to support that many more children index pages. Therefore a given level of the index B-Tree can support more index pages. The more pages are linked from each level to the next, and finally, the fewer numbers of steps from the top page of the index to reach the leaf.

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 to 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 (like an integer, which is just 4 bytes) 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.

WOW! eBook
www.wowebook.org

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.

This is just a general overview of what an index is, and there are several variations of types of indexes in use with SQL Server. Many use a B-Tree structure for their basis, but some use hashing, and others use columnar structures. The most important aspect to understand at this point? Indexes speed access to rows by giving you quicker access to some part of the table so that you don't have to look at every single row and inspect it individually.

On-Disk Indexes

To understand indexes on on-disk objects, it helps to have a working knowledge of the physical structures of a database. At a high level, the storage component of the on-disk engine works with a hierarchy of structures, starting with the database, which is broken down into filegroups (with one `PRIMARY` filegroup always existing), and each filegroup containing a number of files. As we discussed in Chapter 8, a filegroup can contain files for filestream, which in-memory OLTP also uses; but we are going to keep this simple and just talk about simple files that store basic data. This is shown in Figure 10-2.

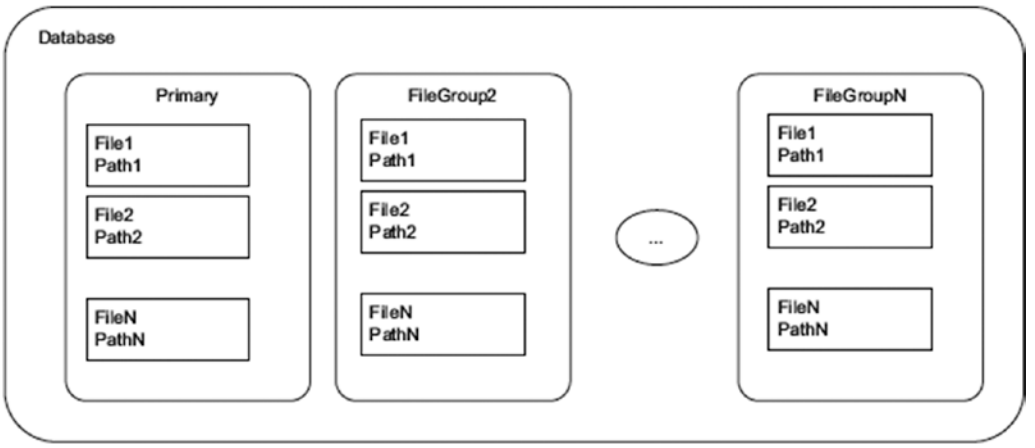


Figure 10-2. *Generalized database and filegroup structure*

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>
```


WOW! eBook
www.wowebook.org

This command assigns the table to the filegroup, but not to any particular file. There are commands to place indexes and constraints that are backed with unique indexes on a different filegroup as well. An important part of tuning can be to see if there is any pressure on your disk subsystems and, if so, possibly redistribute data to different disks using filegroups. If you want to see what files you have in your database, you can query the `sys.filegroups` catalog view:

```
USE <databaseName>;
GO
SELECT CASE WHEN fg.name IS NULL
                --other, such as logs
                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 AS fg
         RIGHT JOIN sys.database_files AS df
            ON fg.data_space_id = df.data_space_id;
```

As shown in Figure 10-3, files are further broken down into a number of extents, each consisting of eight separate 8KB pages where tables, indexes, and so on are physically stored. SQL Server only allocates space in a database in uniques of extents. When files grow, you will notice that the size of files will be incremented only in 64KB increments.

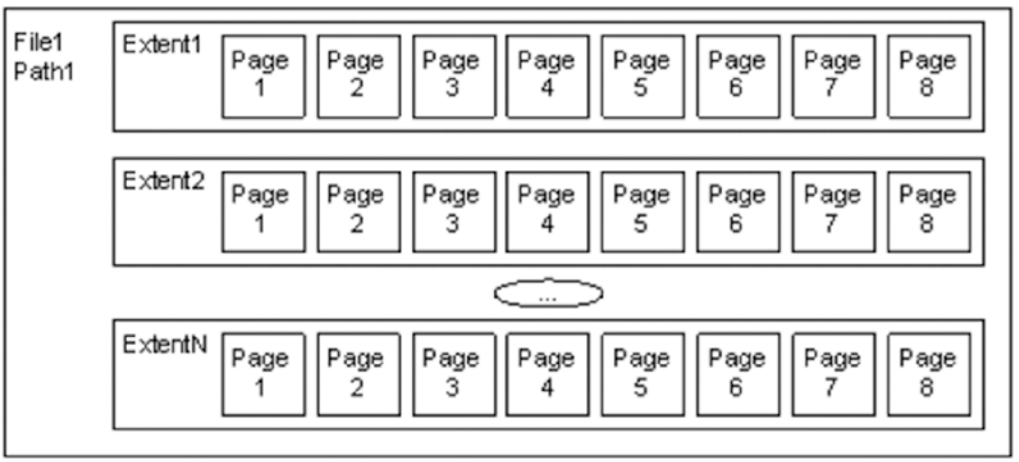


Figure 10-3. *Files and extents*

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.

WOW! eBook
www.wowebook.org

- *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 on-disk 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 is what we typically care about as programmers.

Figure 10-4 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.

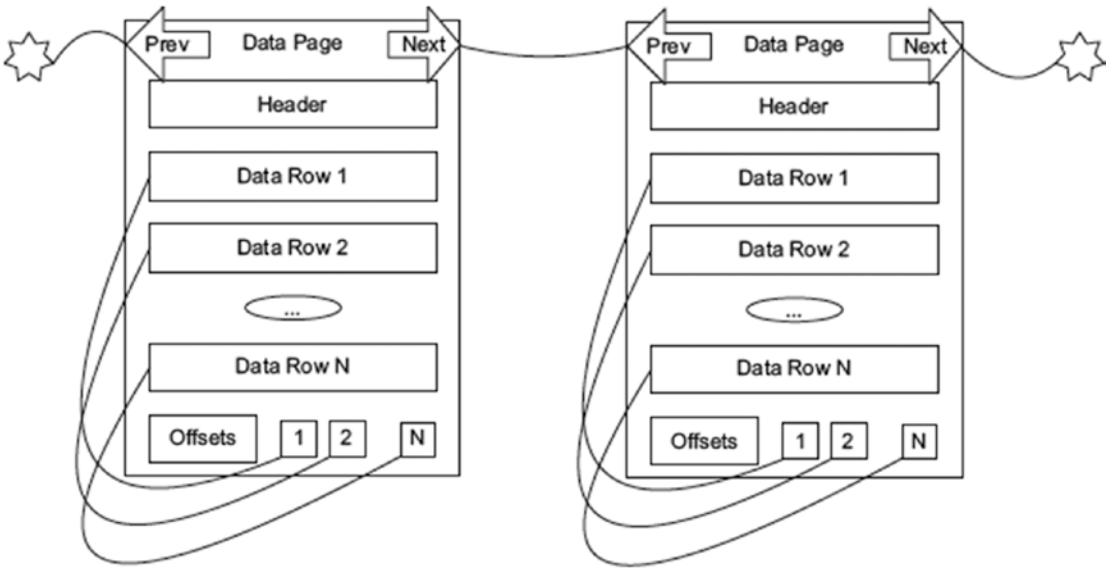


Figure 10-4. *Data pages*

Figure 10-4 also shows that there are pointers from the next to the previous rows. These pointers are added 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 in the next two sections.

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 this case, data goes on an overflow page automatically, allowing you to have virtually unlimited row sizes (with the obvious performance concerns that go along with that, naturally).

WOW! eBook
www.wowebook.org

- 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 10-5 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 support two `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.

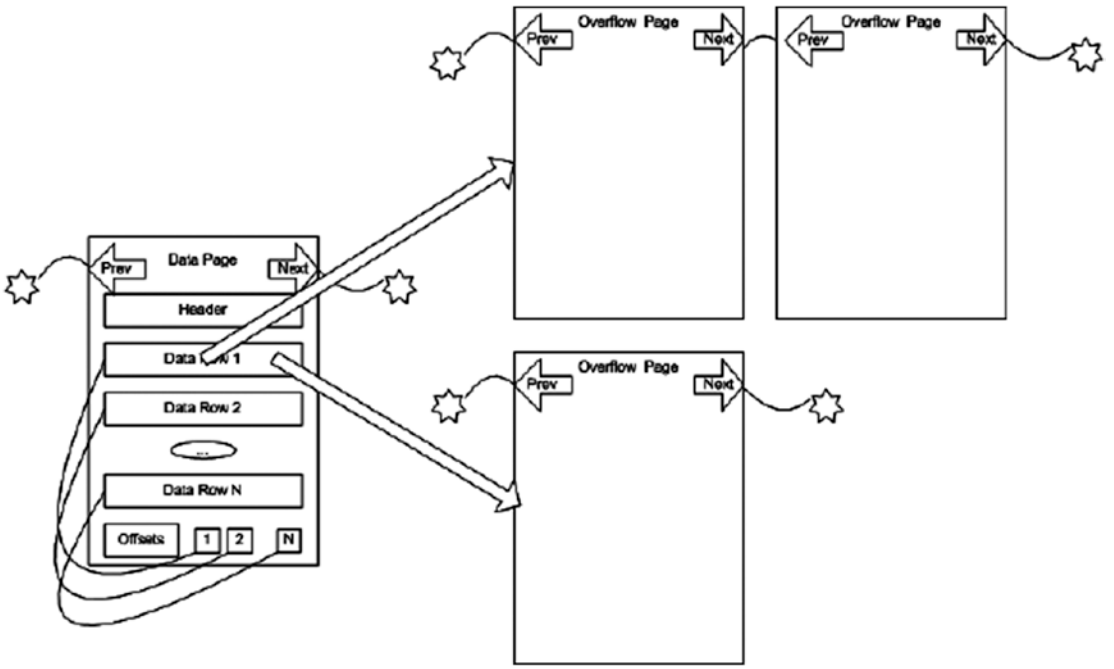


Figure 10-5. *Sample overflow pages*

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.

Storing large values that are placed off of the main page will be far costlier 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'll have to be redirected to the overflow page for every row that's overflowed.

WOW! eBook
www.wowebook.org

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 10-6. (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.)

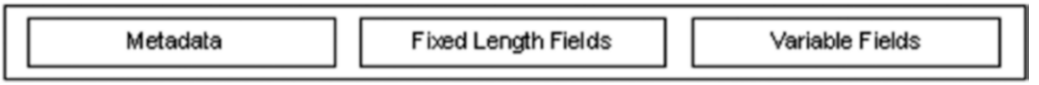


Figure 10-6. *Data row*

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.

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 10-5, when a data row grows larger than 8,060 bytes, the data in variable length columns 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.

One last concept we need to discuss, and that is 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 10-7, assuming that only three values can fit on a page.

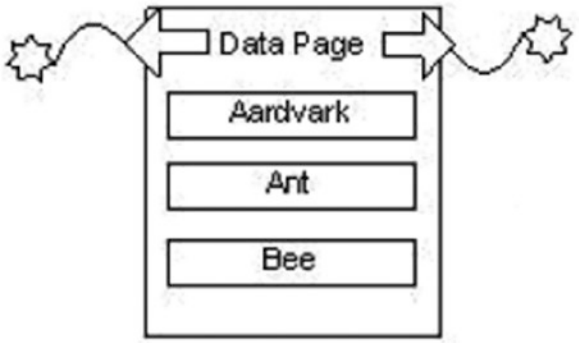


Figure 10-7. *Sample data page before page split*

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% of the data on one page, and 50% 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 10-8.

WOW! eBook
www.wowebook.org

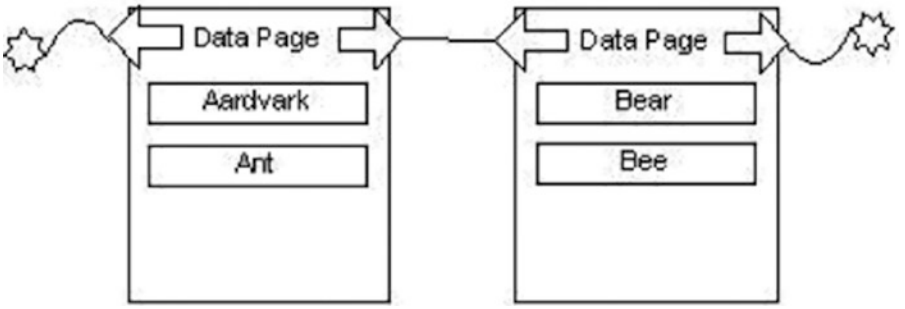


Figure 10-8. *Sample data page after page split*

Page splits are 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 number 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.

Now that we have looked at the basic physical structures, let's get down to the index structures that we will commonly work with. There are two different types of indexes:

- *Clustered*: This type orders the physical table in the order of the index.
- *Nonclustered*: These are completely separate structures that simply speed access.

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. In the upcoming sections, I'll discuss how the different types of indexes affect the table structure and which is best in which situation.

The examples in this section on indexes will mostly be based on tables from the `WideWorldImporters` database you can download from Microsoft in some manner (as I am writing this chapter, it was located on GitHub).

Clustered Indexes

In the following sections, I will discuss the structure of a clustered index, followed by showing the usage patterns and examples of how these indexes are used.

WOW! eBook
www.wowebook.org

Structure

A clustered index physically orders the pages of the data in the table by making the leaf pages of the index be 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. Hence, the records in the physical structure are sorted according to the fields that correspond to the columns used in the index. Tables with a clustered index are referred to as *clustered tables*.

The key of a clustered index is referred to as the *clustering key*. 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 define the key columns of the clustered index on column(s) where the values are unique, and the smaller the better, as the clustering key will be employed in every other index that you place on a table that has a clustered index.

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.)

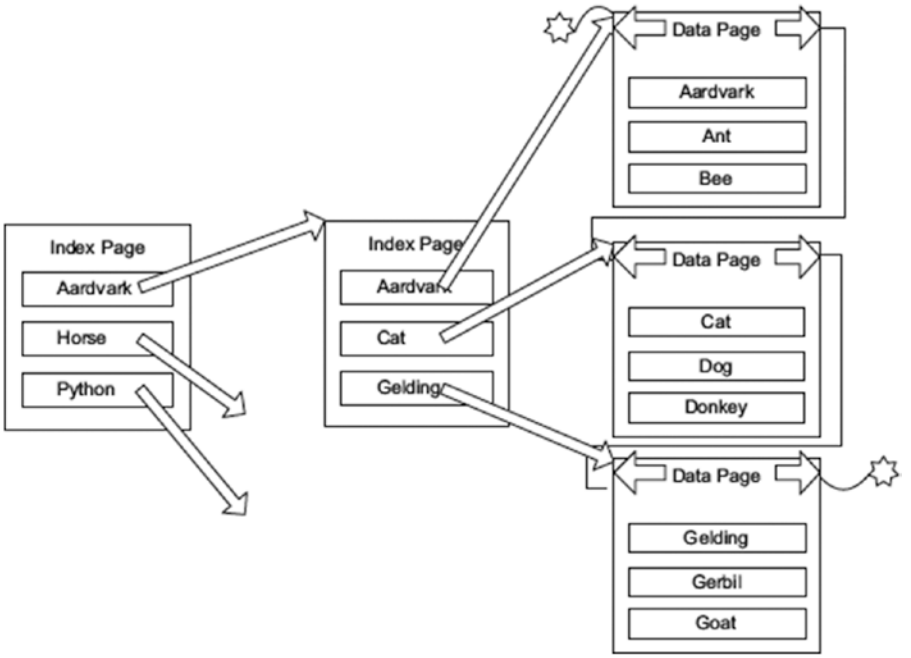


Figure 10-9. *Clustered index example*

You can have only one clustered index on a table, because the table cannot be ordered on more than one set of columns. (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 follow-up questioning.)

A good real-world example of a clustered index would be a set of old-fashioned encyclopedias. Each book 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. These books are clustered on

WOW! eBook
www.wowebook.org

the topics in the encyclopedia, just as the example is clustered on the name of the animal. In essence, the entire set of books is a table of information in clustered order. And indexes can be partitioned as well. The encyclopedias 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 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. 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 `ORDER BY` clauses to make sure that data is returned as you want.

Using the Clustered Index

As mentioned earlier, the clustering key you choose has implications for the rest of your physical design. The column you use for the clustered index will 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 it is a small key (most often, the datatype is an integer that requires only 4 bytes or possibly less using compression), but also 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. GUIDs are random values, in that they generally aren't monotonically increasing, and a new GUID could sort anywhere in a list of other GUIDs and end up causing large amounts of page splits. 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, and does not guarantee sequential order with existing data after a reboot. 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. In SQL Server 2012 and later, the use of the `SEQUENCE` object to generate guaranteed unique values could be used in lieu of GUIDs.

WOW! eBook
www.wowebook.org

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 for which you often need to get a range, such as from A to F.
- *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.

The choice of how to pick the clustered index depends on several 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 unique identifier (GUID) 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 (DMV) `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 select all of the rows in a clustered table, you'll see a `Clustered Index Scan` in the plan. For this we will use the `Application.Cities` table from `WideWorldImporters`, with the following structure:

```
CREATE TABLE Application.Cities
(
    CityID int NOT NULL CONSTRAINT PK_Application_Cities PRIMARY KEY CLUSTERED,
    CityName nvarchar(50) NOT NULL,
    StateProvinceID int NOT NULL
        CONSTRAINT FK_Application_Cities_StateProvinceID_Application_StateProvinces
                REFERENCES Application.StateProvinces (StateProvinceID),
    Location geography NULL,
    LatestRecordedPopulation bigint NULL,
    LastEditedBy int NOT NULL,
    ValidFrom datetime2(7) GENERATED ALWAYS AS ROW START NOT NULL,
    ValidTo datetime2(7) GENERATED ALWAYS AS ROW END NOT NULL,
    PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo)
) ON USERDATA TEXTIMAGE_ON USERDATA
WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE = Application.Cities_Archive))
```


WOW! eBook
www.wowebook.org

This table has temporal extensions added to it, as covered in Chapter 8. In this version of the example database, there is currently also an index on the StateProvince column, which we will use later in the chapter.

```
CREATE NONCLUSTERED INDEX FK_Application_Cities_StateProvinceID ON Application.Cities
(
        StateProvinceID ASC
)
ON USERDATA;
```

There are 37,940 rows in the Application.Cities table:

```
SELECT *
FROM   [Application].[Cities];
```

The plan for this query is as follows:

```
    |--Clustered Index Scan
           (OBJECT:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]))
```

If you query on a value of the clustered index key, the scan will likely change to a seek, and almost definitely if it is backing the PRIMARY KEY constraint. 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   Application.Cities
WHERE  CityID = 23629; --A favorite city of mine, indeed.
```

The plan for this query now does a seek:

```
  |--Clustered Index Seek
          (OBJECT:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]),
           SEEK:([WideWorldImporters].[Application].[Cities].[CityID]=
                                          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 CityID = 23629. 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 parameterize more complex queries as well. (For more information, look up "Simple Parameterization and Forced Parameterization" in Books Online.)

WOW! eBook
www.wowebook.org

You can eliminate the CONVERT_IMPLICIT by explicitly casting the value in your WHERE as WHERE CityID = CAST(23629 AS int):

```
  |--Clustered Index Seek
          (OBJECT:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]),
           SEEK:([WideWorldImporters].[Application].[Cities].[CityID]=[@1])
                                                                    ORDERED FORWARD)
```

Though this not typically done when it is a complementary literal type in the WHERE clause, it can be an issue when you use noncomplimentary types, like when mixing a Unicode value and non-Unicode value, which we will see later in the chapter. Increasing the complexity, now, we search for two rows:

```
SELECT *
FROM   Application.Cities
WHERE  CityID IN (23629,334);
```

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:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]),
           SEEK:([WideWorldImporters].[Application].[Cities].[CityID]=(334)
                 OR
                 [WideWorldImporters].[Application].[Cities].[CityID]=(23629))
                                                                    ORDERED FORWARD)
```

Note that it did not create a parameterized plan this time, but a fixed one with literals for 334 and 23629. Also 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   [Application].[Cities]
WHERE  CityID IN (23629,334);
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:

```
Table 'Cities'. 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 on nonclustered indexes.

WOW! eBook
www.wowebook.org

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 this book. For on-disk tables, nonclustered indexes are completely separate from the data. Instead of the leaf page having all of the data, they have just the index keys (and included values, there are pointers to go to the data pages much like the index of a book contains page numbers).

Each leaf page in a nonclustered index contains some form of link to the rows on the data page. The link from the index to a data row is known as a *row locator*. Exactly how the row locator of a nonclustered index is structured is based on whether or not the underlying table has a clustered index.

In this section, I will start with the structure of nonclustered indexes, and then look at how these indexes are used.

Structure

The base B-tree of the nonclustered index is very much the same as the B-tree of the clustered index. The difference will be how you get to the actual data. First we will look at 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 basic form shown in Figure 10-10.

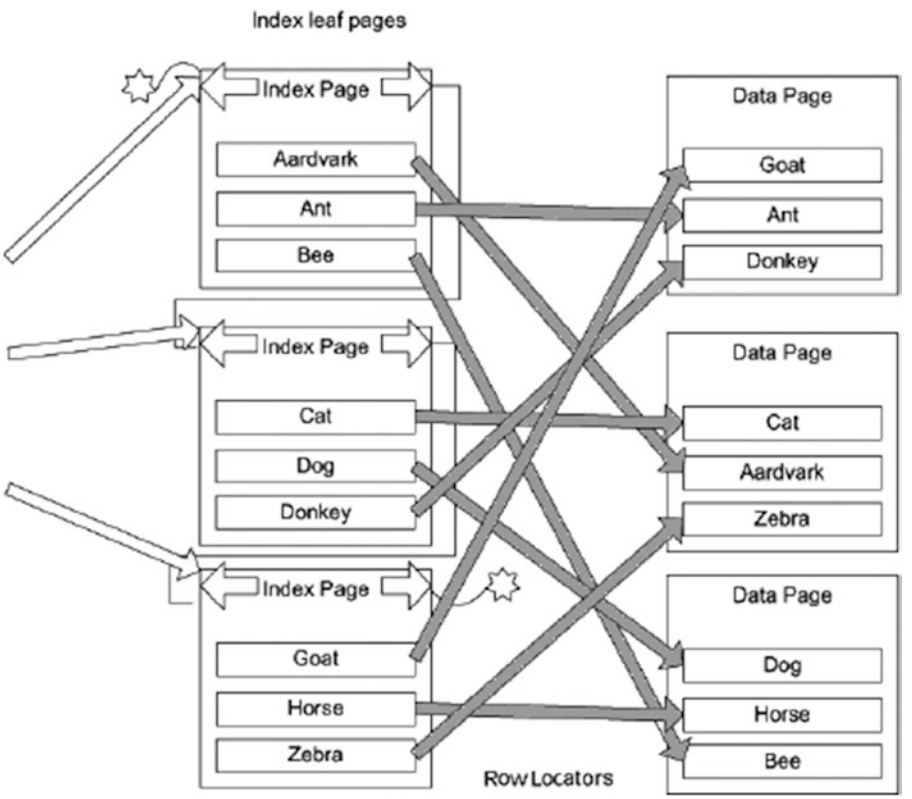


Figure 10-10. *Sample nonclustered index*

WOW! eBook
www.wowebook.org

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, commonly referred to as a row identifier (RID)

In the next two sections, I'll explain these in more detail.

■ **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 Index on a Clustered Table

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 10-11, the structure on the right side represents the clustered index, and the structure on the left represents the nonclustered index. To find a value, you start at the leaf node of the nonclustered index and traverse to the leaf pages. The result of the index traversal is one or more clustering keys, which you then use to traverse the clustered index to reach the data.

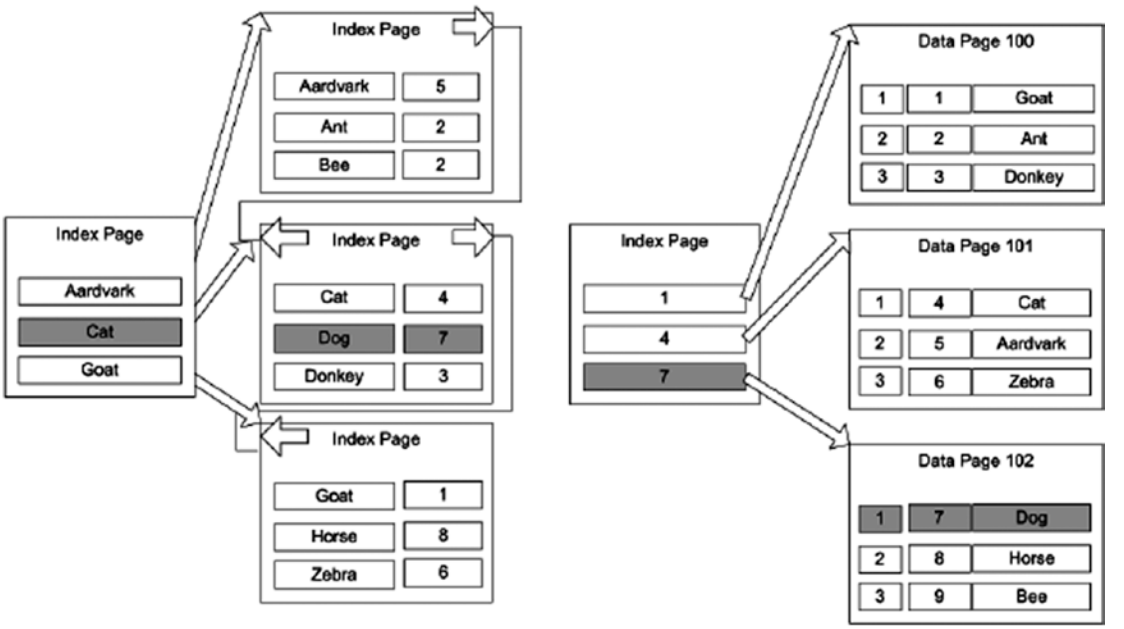


Figure 10-11. *Nonclustered index on a clustered table*

WOW! eBook
www.wowebook.org

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 and subsequently 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 when built this way, data movement is easy for the query processor, and maintaining indexes is an easy operation. Early versions of SQL Server always used physical location pointers for indexes, and this led to corruption in our indexes and tables (without a clustering key, it still uses pointers, but in a manner that reduces corruption at the cost of some performance). Let's face it, the people 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 levels of the clustered index need to 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

A *heap* data structure in computer science is a generally unordered binary tree structure. In SQL Server, when a table does not have a clustered index, the table is physically referred to as a heap. A more practical 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 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.

WOW! eBook
www.wowebook.org

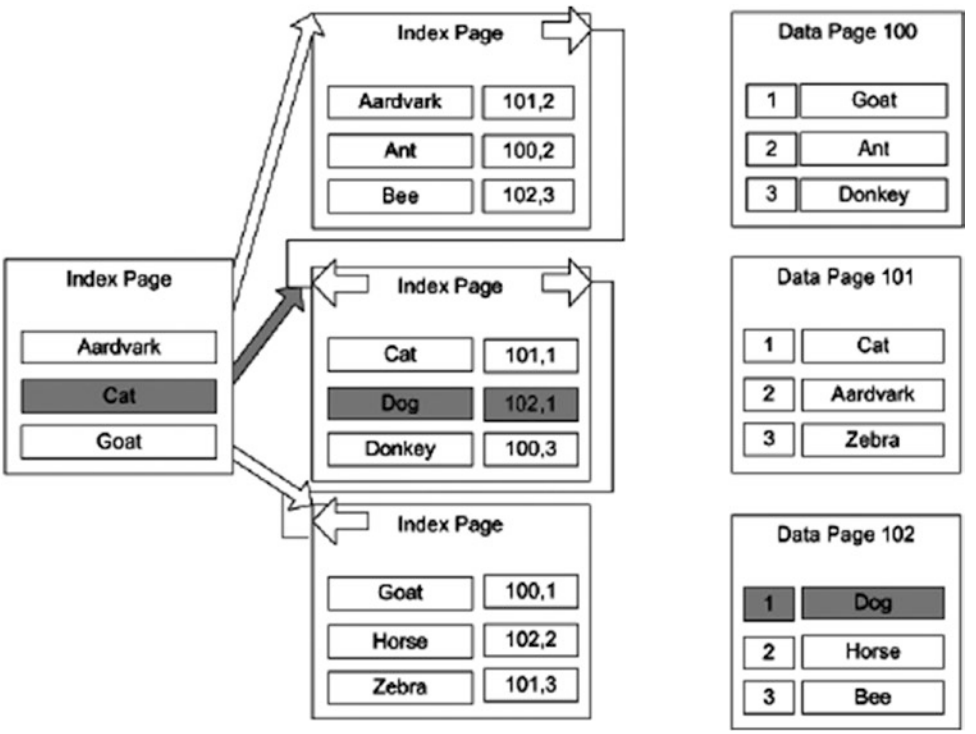


Figure 10-12. *Nonclustered index on a heap*

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: they never change until you rebuild the table. 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, perhaps 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.

WOW! eBook
www.wowebook.org

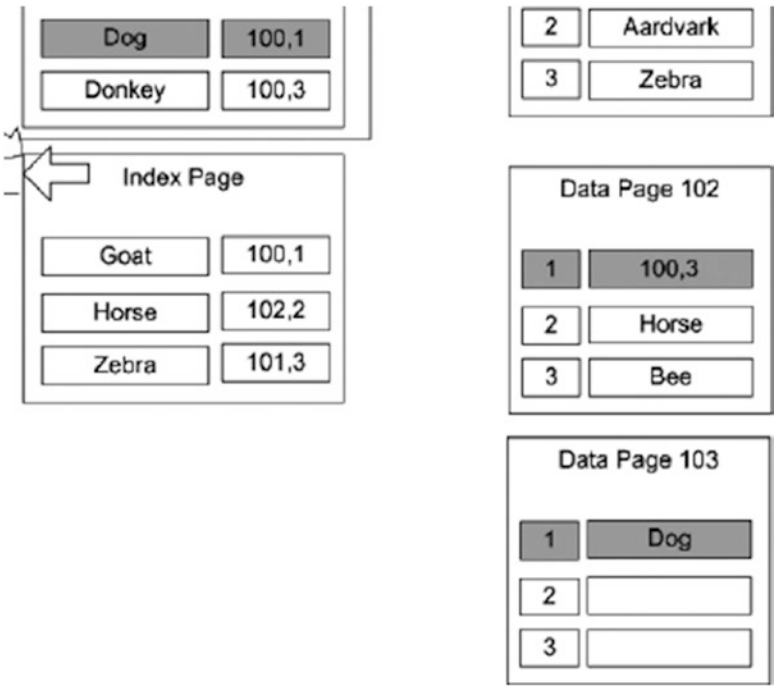


Figure 10-13. *Forwarding pointer*

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, because 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, adds another read to the database. This forwarding pointer is immediately followed when scanning the table, eventually causing possible horrible performance over time if it's not managed.

Space is typically not reused in the heap without rebuilding the table (either by selecting into another table or by using the `ALTER TABLE` command with the `REBUILD` option). In the section "Indexing Dynamic Management View Queries" later in this chapter, 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.

Using the Nonclustered Index

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 some of the choices of how to apply nonclustered indexes in the following areas:

- General considerations
- Composite index considerations
- Nonclustered indexes with clustered tables
- Nonclustered indexes on heaps

WOW! eBook
www.wowebook.org

In reality, many of the topics in this section pertain to clustered indexes—things such as composite indexes, statistics, uniqueness, etc., for certain. I am covering them here because you will typically make these decisions for nonclustered indexes, and choose the clustered index based on patterns of usage as the `PRIMARY KEY`, though not always. There is a section in the last major section of the chapter where I will discuss this in more detail.

General Considerations

We generally start to get the feeling that indexes are needed because queries are (or seem) slow. Naturally though, 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
- Poorly designed database
- Concurrency configuration issues (as covered in Chapter 11)

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 Extended Events 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 need to use that information to consider where to 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 (if usually only slightly). As data is created in the table, there's a chance that the indexes will have to be modified and reorganized to accommodate the new values.
- `UPDATE`: An update physically requires two or three steps: find the row(s) and change the row(s), or find the row(s), delete it (them), and reinsert it (them). During the phase of finding the row, the index is beneficial, just 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 the existing page, or will it require a page split?

WOW! eBook
www.wowebook.org

- 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 start adding nonclustered indexes 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 second 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 all operations that 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 CONCAT(OBJECT_SCHEMA_NAME(i.object_id),'.',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 AS i
         LEFT OUTER JOIN sys.dm_db_index_usage_stats AS s
              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 object_name, index_name;
```

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. A particularly interesting thing to look at in an OLTP system is the user_scans on a clustered index. For example, thinking back to our queries of the Application.Cities table, we only have a clustered index on the PRIMARY KEY column CityID. So if we query

```
SELECT *
FROM   Application.Cities
WHERE  CityName = 'Nashville';
```


WOW! eBook
www.wowebook.org

the plan will be a scan of the clustered index:

```
|--Clustered Index Scan
          (OBJECT:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]),
           WHERE:([WideWorldImporters].[Application].[Cities].[CityName]=(N'Nashvile'));
```

And you will see the `user_scans` value increase every time you run this query. This is something we can reconcile by adding a nonclustered index to this column, locating it in the `USERDATA` filegroup, since that is where other objects were created in the DDL I included earlier:

```
CREATE INDEX CityName ON Application.Cities(CityName) ON USERDATA;
```

Of course, there are plenty more settings when creating an index, some we will cover, some we will not. If you need to manage indexes, it certainly is a great idea to read the Books Online topic about `CREATE INDEX` as a start for more information.

■ **Tip** You can create indexes inline with the `CREATE TABLE` statement starting with SQL Server 2014.

Now check the plan of the query again, and you will see:

```
|--Nested Loops(Inner Join,
              OUTER REFERENCES:([WideWorldImporters].[Application].[Cities].[CityID]))
       |--Index Seek(OBJECT:([WideWorldImporters].[Application].[Cities].[CityName]),
            SEEK:([WideWorldImporters].[Application].[Cities].[CityName]=N'Nashville')
                                                                    ORDERED FORWARD)
       |--Clustered Index Seek
             (OBJECT:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]),
              SEEK:([WideWorldImporters].[Application].[Cities].[CityID]=
               [WideWorldImporters].[Application].[Cities].[CityID])
                                                  LOOKUP ORDERED FORWARD)
```

This not only is more to format, but it looks like it should be a lot more to execute. However, this plan illustrates nicely the structure of a nonclustered index on a clustered table. The second `|--` points at the improvement. To find the row with `CityName` of Nashville, it seeked in the new index. Once it had the index keys for Nashville rows, it took those and joined them together like two different tables, using a Nested Loops join type. For more information about nested loops and other join operators, check Books Online for "Showplan Logical and Physical Operators Reference."

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, then things will improve. There's a bit of truth to this in a lot of cases, but indexes have to be useful to be used by a query. What if the index of a 418-page book has two entries:

General Topics 1

Determining Index Usefulness 417

WOW! eBook
www.wowebook.org

This means that pages 1–416 cover general topics and pages 417-? are about determining index usefulness. This would be useless to you, unless you needed to know about index usefulness. One thing is for sure: you could determine that the index is generally useless pretty quickly. Another thing we all do with the index of a book to see if the index is 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 the index we just created:

```
DBCC SHOW_STATISTICS('Application.Cities', 'CityName') WITH DENSITY_VECTOR;
DBCC SHOW_STATISTICS('Application.Cities', 'CityName') WITH HISTOGRAM;
```

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
------------- -------------- -------------------------
4.297009E-05  17.17427       CityName
2.635741E-05  21.17427       CityName, CityID

RANGE_HI_KEY           RANGE_ROWS    EQ_ROWS       DISTINCT_RANGE_ROWS  AVG_RANGE_ROWS
---------------------- ------------- ------------- -------------------- --------------
Aaronsburg             0             1             0                    1
Addison                123           11            70                   1.757143
Albany                 157           17            108                  1.453704
Alexandria             90            13            51                   1.764706
Alton                  223           13            122                  1.827869
Andover                173           13            103                  1.679612
.........                   ...           ..            ...                  ........
White Oak              183           10            97                   1.886598
Willard                209           10            134                  1.559701
Winchester             188           18            91                   2.065934
Wolverton              232           1             138                  1.681159
Woodstock              137           12            69                   1.985507
Wynnewood              127           2             75                   1.693333
Zwolle                 240           1             173                  1.387283
```

I won't cover the `DBCC SHOW_STATISTICS` command in great detail, but there are several important things to understand. First, consider the density of each column set. The `CityName` 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 clustering key as well.

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)
```


WOW! eBook
www.wowebook.org

```
SELECT 1.0/ COUNT(DISTINCT ISNULL(CityName,'NotACity')) AS density,
            COUNT(DISTINCT ISNULL(CityName,'NotACity')) AS distinctRowCount,
            1.0/ COUNT(*) AS uniqueDensity,
            COUNT(*) AS allRowCount
FROM   Application.Cities;
```

This returns the following:

```
density              distinctRowCount uniqueDensity     allRowCount
-------------------- ---------------- ----------------- -----------
0.000042970092       23272            0.000026357406    37940
```

You can see that the densities match. (The query's 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.

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
------------ ------------- ------------- -------------------- --------------
Aaronsburg   111           58            2                    55.5
Addison      117           67            2                    58.5
...          ...           ...           ...                  ...
```

In the second row, the row values tell us the following:

- `RANGE_HI_KEY`: The sampled `CityName` values are `Aaronsburg` and `Addison`.
- `RANGE_ROWS`: There are 117 rows where the value is between `Aaronsburg` and `Addison` (noninclusive of the endpoints). These values would not be known. However, if a user uses `Aaronsville` as a search argument, the optimizer can now guess that a maximum of 117 rows would be returned (stats are not kept up as part of the transaction). 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 are exactly 67 rows where `CityName = Addison`.
- `DISTINCT_RANGE_ROWS`: For the row with `Addison`, it is estimated that there are two distinct values between `Aaronsburg` and `Addison`.
- `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`.

WOW! eBook
www.wowebook.org

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 (using tempdb):

```
USE tempDB;
GO
CREATE SCHEMA demo;
GO
CREATE TABLE demo.testIndex
(
    testIndex int IDENTITY(1,1) CONSTRAINT PKtestIndex PRIMARY KEY,
    bitValue bit,
    filler char(2000) NOT NULL DEFAULT (REPLICATE('A',2000))
);
CREATE INDEX bitValue ON demo.testIndex(bitValue);
GO

SET NOCOUNT ON; --or you will get back 50100 1 row affected messages
INSERT INTO demo.testIndex(bitValue)
VALUES (0);
GO 50000 --runs current batch 50000 times in Management Studio.

INSERT INTO demo.testIndex(bitValue)
VALUES (1);
GO 100 --puts 100 rows into table with value 1
```

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   demo.testIndex
WHERE  bitValue = 0;
```

This shows a clustered index scan:

```
|--Clustered Index Scan(OBJECT:([tempdb].[demo].[testIndex].[PKtestIndex]),
                        WHERE:([tempdb].[demo].[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].[ demo].[testIndex].[testIndex], [Expr1003]) WITH UNORDERED PREFETCH)
    |--Index Seek(OBJECT:([tempdb].[demo].[testIndex].[bitValue]),
                  SEEK:([tempdb].[demo].[testIndex].[bitValue]=(1)) ORDERED FORWARD)
    |--Clustered Index Seek(OBJECT:([tempdb].[demo].[testIndex].[PKtestIndex]),
             SEEK:([tempdb].[demo].[testIndex].[testIndex]=
                      [tempdb].[demo].[testIndex].[testIndex]) LOOKUP ORDERED FORWARD)
```


WOW! eBook
www.wowebook.org

As we saw earlier, this better plan looks more complicated, but it hinges on now only needing to touch a few 100 rows, instead of 50,100 in the Index Seek operator because we chose to do `SELECT *` (more on how to avoid the clustered seek in the next section).

You can see why in the histogram:

```
UPDATE STATISTICS demo.testIndex;
DBCC SHOW_STATISTICS('demo.testIndex', 'bitValue')  WITH HISTOGRAM;
```

This returns the following results in my test. Your actual values will likely vary.

```
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 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 simple demonstration of the histogram is one thing, but in practice, actually building a filtered index to optimize this query is a generally better practice. You might build an index such as this:

```
CREATE INDEX 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 another index would perform, which can also be dependent on a myriad of other SQL Server internals. 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).

WOW! eBook
www.wowebook.org

As we discussed in the "Clustered Indexes" 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 on single columns, but it isn't always that you need performance-enhancing indexes only 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 first, then the second column. So the second column in the index is generally only useful if you need the first column as well (the next section on covering indexes demonstrates a way that this may not be the case, however). Even so, you will very often need composite indexes to optimize common queries when predicates on all of the columns are involved.

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 likely 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 (`StateProvince` is a more obvious choice of column, but it has an index that we will be using in a later section):

```
SELECT *
FROM   Application.Cities
WHERE  CityName = 'Nashville'
  AND  LatestRecordedPopulation = 601222;
```

The index on `CityName` we had is useful, but an index on `LatestRecordedPopulation` might also be good. It may also turn out that neither column alone provides enough of an improvement in performance. 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 `CityName = 'Nashville'` and `LatestRecordedPopulation = 601222`.

WOW! eBook
www.wowebook.org

The preceding query with existing indexes (clustered primary key on CityId, indexes on StateProvinceId that was part of the original structure, and CityName) is optimized with the previous plan, with the extra population predicate:

```
  |--Nested Loops(Inner Join,
              OUTER REFERENCES:([WideWorldImporters].[Application].[Cities].[CityID]))
       |--Index Seek(OBJECT:([WideWorldImporters].[Application].[Cities].[CityName]),
              SEEK:([WideWorldImporters].[Application].[Cities].[CityName]=N'Nashville')
                                                                        ORDERED FORWARD)
       |--Clustered Index Seek
             (OBJECT:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]),
     SEEK:([WideWorldImporters].[Application].[Cities].[CityID]=
                                  [WideWorldImporters].[Application].[Cities].[CityID]),
WHERE:([WideWorldImporters].[Application].[Cities].[LatestRecordedPopulation]=(601222))
                                                                  LOOKUP ORDERED FORWARD)
```

Adding an index on CityName and LatestRecordedPopulation 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 CityName, LatestRecordedPopulation, COUNT(*) AS [count]
FROM   Application.Cities
GROUP BY CityName, LatestRecordedPopulation
ORDER BY CityName, LatestRecordedPopulation;
```

This returns partial results:

```
CityName         LatestRecordedPopulation count
---------------- ------------------------ -----------
Aaronsburg       613                      1
Abanda           192                      1
Abbeville        419                      1
Abbeville        2688                     1
Abbeville        2908                     1
Abbeville        5237                     1
Abbeville        12257                    1
Abbotsford       2310                     1
Abbott           NULL                     2
Abbott           356                      3
Abbottsburg      NULL                     1
```

Of course, you can't always look at all of the rows like this, so another possibility is to do a little data profiling to see which of the columns has more distinct values, and look for NULL values when columns are nullable:

```
SELECT COUNT(DISTINCT CityName) as CityName,
       SUM(CASE WHEN CityName IS NULL THEN 1 ELSE 0 END) as NULLCityName,
       COUNT(DISTINCT LatestRecordedPopulation) as LatestRecordedPopulation,
```


WOW! eBook
www.wowebook.org

```
       SUM(CASE WHEN LatestRecordedPopulation IS NULL THEN 1 ELSE 0 END)
                                                              AS NULLLatestRecordedPopulation
FROM   Application.Cities;
```

This query returns the following:

```
CityName    NULLCityName LatestRecordedPopulation NULLLatestRecordedPopulation
----------- ------------ ------------------------ ----------------------------
23272       0            9324                     11048
```

The column CityName has the most unique values, which kind of runs counter to what you would expect. However, lots of NULL column values will do that. So we add the following index:

```
CREATE INDEX CityNameAndLastRecordedPopulation
         ON Application.Cities (CityName, LatestRecordedPopulation);
```

Now, reexecute the query we ran before the index was created:

```
SELECT *
FROM   Application.Cities
WHERE  CityName = 'Nashville'
  AND  LatestRecordedPopulation = 601222;
```

The plan changes to the following, using the new index:

```
  |--Nested Loops(Inner Join,
                 OUTER REFERENCES:([WideWorldImporters].[Application].[Cities].[CityID]))
       |--Index Seek
           (OBJECT:([WideWorldImporters].[Application].[Cities].
                                                    [CityNameAndLastRecordedPopulation]),
            SEEK:([WideWorldImporters].[Application].[Cities].[CityName]=N'Nashville'
              AND WideWorldImporters].[Application].[Cities].[LatestRecordedPopulation]
                                                  =(601222)) ORDERED FORWARD)
       |--Clustered Index Seek
            (OBJECT:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]),
             SEEK:([WideWorldImporters].[Application].[Cities].[CityID]
             =[WideWorldImporters].[Application].[Cities].[CityID]) LOOKUP ORDERED FORWARD)
```

Keep in mind, too, exactly how the indexes will be used. If your queries mix equality comparisons 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.

In the next section, I will show how you can eliminate the clustered index seek, but in general, having the seek 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.

WOW! eBook
www.wowebook.org

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.

From our previous examples, if instead of returning all columns in the table, we just returned `CityName` and `LatestRecordedPopulation`:

```
SELECT CityName, LatestRecordedPopulation
FROM   Application.Cities;
```

the resulting plan would not be complex at all—just a simple scan for the rows, using just the index:

```
  |--Index Scan
  (OBJECT:([WideWorldImporters].[Application].[Cities].[CityNameAndLastRecordedPopulation]))
```

But what if we also needed the `LastEditedBy` column in the output. We could add the column to the index as an index key, but if it isn't needed for the search, that is wasteful. Instead, there is a feature of an index 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 generally terrible and very outdated/deprecated anyhow.

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, first, check the plan on the following query:

```
SELECT CityName, LatestRecordedPopulation, LastEditedBy
FROM   Application.Cities;
```

It is a scan through the clustered index:

```
|--Clustered Index Scan
       (OBJECT:([WideWorldImporters].[Application].[Cities].[PK_Application_Cities]))
```

Now let's modify the index on city name and population and include the `LastEditedBy` column:

```
DROP INDEX CityNameAndLastRecordedPopulation
         ON Application.Cities;

CREATE INDEX CityNameAndLastRecordedPopulation
         ON Application.Cities (CityName, LatestRecordedPopulation)
                 INCLUDE (LastEditedBy);
```


WOW! eBook
www.wowebook.org

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 Scan
  (OBJECT:([WideWorldImporters].[Application].[Cities].[CityNameAndLastRecordedPopulation]))
```

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 as you are duplicating data. 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 costliest 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 that are executed frequently. It's almost always better to support a specific 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.

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!)

You 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.

WOW! eBook
www.wowebook.org

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.

As an example, using the table we have been working with, I want to search for the Nashville in StateProvince 44 (Tennessee). There is an index on the `CityName` column, and there is one on the `StateProvince` table as part of the table as it was created. Checking the plan on the following query:

```
SELECT CityName, StateProvinceID --limiting output to make the plan easier to follow
FROM   Application.Cities
WHERE  CityName = 'Nashville'
  AND  StateProvinceID = 44;
```

produces the following plan:

```
  |--Merge Join(Inner Join,
         MERGE:([WideWorldImporters].[Application].[Cities].[CityID])=
                                  ([WideWorldImporters].[Application].[Cities].[CityID]),
         RESIDUAL:([WideWorldImporters].[Application].[Cities].[CityID] =
                                 [WideWorldImporters].[Application].[Cities].[CityID]))
       |--Index Seek(OBJECT:([WideWorldImporters].[Application].[Cities].[CityName]),
                   SEEK:([WideWorldImporters].[Application].[Cities].[CityName]=N'Nashville')
                                                                        ORDERED FORWARD)
       |--Index Seek (OBJECT:([WideWorldImporters].[Application].[Cities].
                                                  [FK_Application_Cities_StateProvinceID]),
                  SEEK:([WideWorldImporters].[Application].[Cities].[StateProvinceID]=(44))
                                                                        ORDERED FORWARD)
```

Looking at the plan for this query, you can see that there are two index seeks to find rows where `CityName = 'Nashville'` and `StateProvince = 44`. 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 `CityName` values in alphabetical order, but then ordered by `LastRecordedPopulation` in descending order:

```
SELECT CityName, LatestRecordedPopulation
FROM   Application.Cities
ORDER BY CityName ASC, LatestRecordedPopulation DESC;
```


WOW! eBook
www.wowebook.org

The plan for this query follows:

```
  |--Sort(ORDER BY:([WideWorldImporters].[Application].[Cities].[CityName] ASC,
           [WideWorldImporters].[Application].[Cities].[LatestRecordedPopulation] DESC))
       |--Index Scan
          (OBJECT:([WideWorldImporters].[Application].[Cities].
                                                    [CityNameAndLastRecordedPopulation]))
```

But change the index to match the sort order that we are using, keeping the INCLUDE column:

```
DROP INDEX CityNameAndLastRecordedPopulation
         ON Application.Cities;

CREATE INDEX CityNameAndLastRecordedPopulation
         ON Application.Cities (CityName, LatestRecordedPopulation DESC)
                 INCLUDE (LastEditedBy);
```

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:

```
|--Index Scan
            (OBJECT:([WideWorldImporters].[Application].[Cities].
                                        [CityNameAndLastRecordedPopulation]), 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 make a copy of the table we have been working with, and make the primary key a nonclustered one:

```
SELECT *
INTO   Application.HeapCities
FROM   Application.Cities;

ALTER TABLE Application.HeapCities
   ADD CONSTRAINT PKHeapCities PRIMARY KEY NONCLUSTERED (CityID);

CREATE INDEX CityName ON Application.HeapCities(CityName) ON USERDATA;
```


WOW! eBook
www.wowebook.org

Now, we look for a single value in the table:

```
SELECT *
FROM   Application.HeapCities
WHERE  CityID = 23629;
```

The following plan will be used to execute the query:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([Bmk1000]))
     |--Index Seek
         (OBJECT:([WideWorldImporters].[Application].[HeapCities].[PKHeapCities]),
          SEEK:([WideWorldImporters].[Application].[HeapCities].[CityID]=
                                          CONVERT_IMPLICIT(int,[@1],0)) ORDERED FORWARD)
     |--RID Lookup(OBJECT:([WideWorldImporters].[Application].[HeapCities]),
          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 RID lookup operator is the most important thing I wanted to show in this section, so you can identify this on a plan and understand what is going on. This RID lookup operator is very similar to 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.

Memory-Optimized Indexes

In SQL Server 2012, Microsoft started implementing what have been known as memory-optimized indexes with read-only nonclustered columnstore indexes. These indexes use a columnar approach to data storage that, instead of storing rows together, store each column together. This is further enhanced by an engine called xVelocity that provides excellent compression, and enhanced performance for some uses.

WOW! eBook
www.wowebook.org

In 2014, while enhancing columnstore indexes to include a writable clustered version (though without the ability to have additional row-based indexes), Microsoft added something else that we have alluded to many times already in this book: In-Memory OLTP (which was also known as Hekaton…a term that you may see when doing searches about this technology). It was very limited at a table level (no check or foreign key constraints, only a single uniqueness constraint, etc.) and worked primarily for cases where an application treats SQL Server as just an optimized data storage bucket.

Finally (as far as this edition of this book goes), in 2016, Microsoft enhanced both technologies, allowing nonclustered indexes on columnstore clustered tables, allowing updateable nonclustered columnstore indexes, and giving in-memory OLTP tables check constraints, FOREIGN KEY constraints between in-memory tables, and up to eight uniqueness indexes/constraints. All this to say…memory-optimized technologies are fairly new, very specific in what they are used for, and changing very fast.

In this section, I will give you an overview of the structures and usage for these technologies, just like I did the on-disk versions, but the story here is in the process of being created, and is very likely to change faster than I could provide you with new editions of a book. With the new Azure paradigms, in-memory and columnstore-based technologies could change 20 times before even the next box edition comes out. So stay tuned and hang on. (As I make final edits, some limitations have been removed for the next release already in CTP!)

In this section, I will start with an overview of in-memory OLTP tables and indexes, and then provide an overview of the basic structure of a columnstore index. For the columnstore index, I will only provide some base examples, and will leave that discussion to Chapter 14, where reporting is covered, since columnstore indexes are much more aligned to either reporting databases, or a few new reporting scenarios in 2016 that allow you to report right off of your OLTP database using an asynchronously maintained index. The discussion will be mostly in terms of the 2016 release, so if you have something earlier or later, it would do you well to check for changes in details. The paradigms will not change much most likely, but features definitely will.

One major difference between on-disk and in-memory indexes is that in-memory indexes are not logged and are not stored anywhere. When you shut down the server, they are gone. When you restart the server, the indexes are rebuilt as the data is read in from the disk files that back up the memory structures. As such, changes to in-memory tables do not log changes to the indexes.

In-Memory OLTP Tables

While programming with in-memory tables is very much the same T-SQL code you already know, internally things are very different. Data resides in-memory as its home, rather than on disk, and while there is still an internal record size limitation of 8K, memory is not organized to match on-disk files. Data is persisted to disk in a very similar pattern to on-disk tables. When you make a change to data, the changes are written to memory and to the log (if you have specified the table to be durable), and before the transaction completes, data is written to the transaction log (unless you have `DELAYED_DURABILITY` enabled for the database).

■ **Tip** You can use temporal extensions with in-memory tables, but the history table will be an on-disk table.

Similarities generally end there. The structure of the rows is different, indexes are very different, as is what happens during an `UPDATE` or `DELETE`. In this section, we will discuss the general structure of in-memory objects and indexes, and then we will take a look at the two types of indexes, including one that is very new to the SQL Server engine.

WOW! eBook
www.wowebook.org

General Table Structure

The structure of the in-memory OLTP objects uses a temporal-style structure that is the basis of Multi-Version Concurrency Control internal structures. Every change to the database results in a new row in a structure, and/or an update to an effective timestamp. Locks and latches are not employed to enable consistent modification of resources, and no connection ever waits for anything other than hardware busy doing work for another connection.

The basic structure of a record in the table is shown in Figure 10-14. Nothing too interesting in the basic structure. However, the row is limited to 8060 bytes, and data that takes more space than this is stored in its own internal table. For most cases, a row size that approaches 8060 bytes may not be great for usage in memory, particularly if you have a lot of rows (be sure and test it out in any case). Where things get interesting is in the Record Header, as shown in Figure 10-15.

Figure 10-14. *Basic row structure*

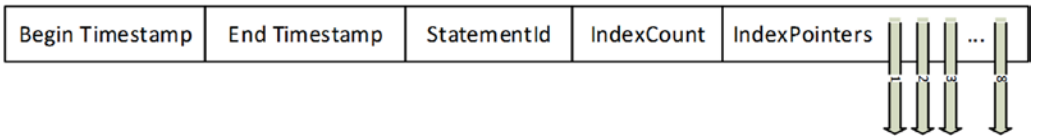


Figure 10-15. *In-Memory Record Header*

The Begin Timestamp and End Timestamp represent a time frame in which the row is valid (Time >= [Begin Timestamp] and Time < [End TimeStamp]). This timestamp is used by transactions that have started to let the connection know what rows in the structure it can see. All concurrency (covered in Chapter 11 in greater detail) is handled using a SNAPSHOT isolation level manner, in that a transaction will always see its distinct view of the database as of the start of its transaction.

As an example, say we have a partial table with two columns, `AddressId`, a unique integer value, and `Country`, with the name of a country. From timestamp 0 – 100, it looked like:

```
AddressId        Country              Value
---------------- -------------------- ------
1                USA                  T
2                USA                  D
3                Canada               D
4                Colombia             D
```

Then, at timestamp 100, it changed to the following, with address 2 now being a Canadian one:

```
AddressId        Country              Value
---------------- -------------------- ------
1                USA                  T
2                Canada               D
3                Canada               D
4                Colombia             D
```


WOW! eBook
www.wowebook.org

However, a connection still has a transaction open at timestamp 50, so the old view of the structure needs to remain. Figure 10-16 shows how this would look (technically, in-memory tables need a unique index, but this is an abstract example of the basic structure).

Start	End	Index	AddessId	Country	Value
0	∞	Φ Φ	1	USA	T
0	100	Φ Φ	2	USA	D
100	∞	Φ Φ	2	Canada	D
0	∞	Φ Φ	3	Canada	D
0	∞	Φ Φ	4	Colombia	D

Figure 10-16. *Sample table structure*

By filtering on the two time frames, you can see one of the rows where `AddressId = 2` falls out, leaving us with the two tables of the data.

The other very different part of the structure is how indexes work. The arrows in Figure 10-17 are index pointers. The part of the structure to find the first row is always a separate structure, but the physical record pointer just points to one row in the table. From there, all values that meet the criteria for a level in the index become a chain of rows.

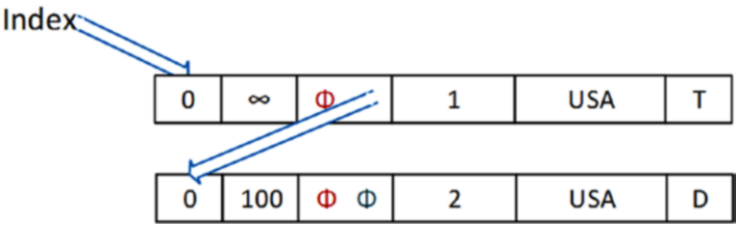


Figure 10-17. *Index pointers*

In the example, one row from an index would point to the first row in the image, and the pointer would be followed for the second row. For the unique index, you instinctively might think there would never be a chain of rows, but this is not the case. For updates, you end up with multiple rows until the old rows are removed. Also, if using a hash index, it does not have uniqueness in the way it is implemented, it still needs to look for uniqueness. For any of the index types, the goal is generally to have no more than an average of 100 values in this chain to be scanned. If this is the case, it will be best to choose your index key columns so they are more unique (this should not be an issue with indexes you are using for a `PRIMARY KEY/UNIQUE` constraint.)

WOW! eBook
www.wowebook.org

The index pointer structure is both limiting and empowering. It is limiting because you can only have the eight indexes. On the other hand, the leaf node of ALL indexes are the table's data. So the primary benefit of a clustered index, that of not having to go fetch the rest of the data from a separate structure, is there for all in-memory OLTP indexes because of how they are structured. The downside is that we have a maximum of eight indexes per table.

Conceptually and structurally, things are very different, but for the most part, assuming you are designing and implementing memory-optimized tables, the most important thing is to understand the basic versioning structure to understand which index to choose when, and to really grok the differences in how concurrency is implemented.

Index Structure

When Microsoft created the in-memory OLTP engine, they didn't just create one type of index like we had in the on-disk structures. There are two types (and you can use columnstore indexes as well). B-Tree based indexes are awesome structures, but they can be overkill when you simply need to do a single-row lookup. So Microsoft has given us (for in-memory objects only, so far) a new index type, a *hash index*.

In this section, we will look at the B-Tree based Bw-Tree indexes, as well as the new hash index types that are used for the in-memory OLTP structures.

Bw-Tree Indexes

The Bw-Tree index is, for all intents and purposes, very similar to the current B-Tree indexes that we have had in SQL Server since the very beginning. In the next two sections, I will look at the conceptual implementation and usage patterns for Bw-Tree indexes. In Figure 10-18, you can see the base structure of the index. There is a page mapping layer that contains the memory addresses of the pages of the index, and a B-Tree structure that conceptually is the same as on on-disk B-Tree, except the lower values of the tree are less than the key on the parent page. So, because Page 0 has C as the first entry, all values on the leftmost node will be C or less.

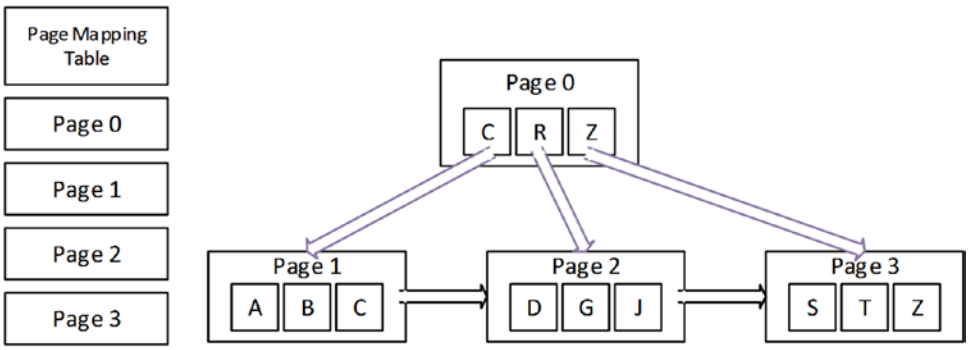


Figure 10-18. *Bw-Tree index node pages*

The values in the nodes of the index are not timestamped, and are maintained for all values. The values on the leaf nodes point to actual rows in the table. In our example table, this node set is an index that includes the `Value` column from the rows in our sample structure, as shown in Figure 10-19.

WOW! eBook
www.wowebook.org

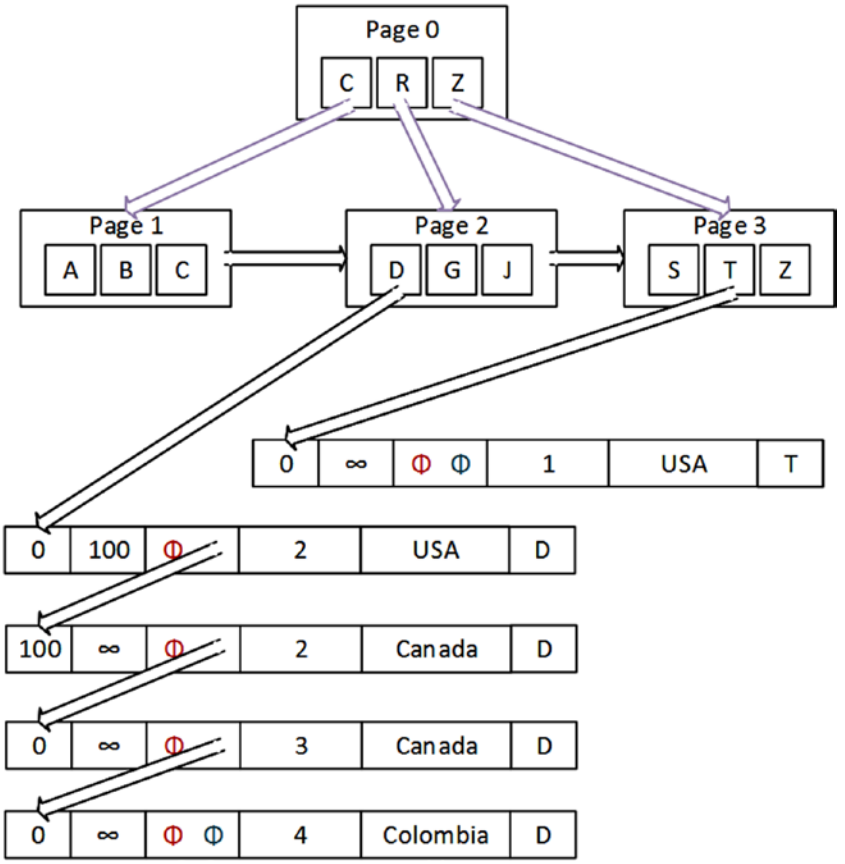


Figure 10-19. *Bw-Tree index node pages linked to data*

As shown in the intro section to memory-optimized tables, once you reach the value in the leaf node, it is the entry point into the internal linked list of index pointers. So all of the D values and versions we have are chained together row by row, and any older transactions are cleaned up as the engine can get to it.

Hash Indexes

Hashing is a well-tested concept in computer science, and has been used by the optimizer since SQL Server 7.0 to do joins on unindexed, unordered sets. The idea is that you create a function that takes (more or less) an infinite number of inputs, and outputs a finite, manageable "bucket" to match them in. For example, suppose you have the following data, and the ellipsis represents approximately 100 rows:

WOW! eBook
www.wowebook.org

```
Name
-------------------
Amanda
Emma
Aria
Andrew
Addison
...
Jim
Linda
Max
```

Say all you ever need to do is find a name in the list. In a Bw-Tree, you would have the overhead of sorting the data, and perhaps also duplicating the data in the leaf pages of the index. In a hash index, you simply devise a function that will let you separate things into piles. As a human, you might use the first letter or letters of the name; or perhaps the length of the name, which gives you the following:

```
Name                HashBucket
------------------- --------------------
Amanda              6
Emma                4
Aria                4
Andrew              6
Addison             7
...
Jim                 3
Linda               5
Max                 3
```

You sort the data by the HashBucket, and then scan the bucket. This is very similar to what SQL Server will do with hash index structures, only with a hash function that will produce a lot greater variety in hash bucket values. In this section, I will look at conceptually how this is applied to hash indexes, and then cover some of the basics of creating and using hash indexes.

SQL Server basically implements hash buckets as one might do manually, using an unknown algorithm to do the hashing that you have no control over. (There is one hash algorithm for all data types.) You simply choose a number of buckets based on the expected number of unique values × approximately 2, which will be internally rounded up to the next highest power of 2. Then create the index and it does the work. Each bucket takes 8 bytes upon creation (or index rebuild), and does not change on its own. You have to resize the bucket count manually if your estimated number of unique values changes. Bear in mind that it is far better to have way too many hash buckets than too few. As stated in the intro section, if too many values are placed in the chain to scan, performance will degrade. The larger the number of hash buckets, the less likely this scenario will become.

As an example, in Figure 10-20, I have the structure that I started with earlier in this section, but now I have a hash index on the `Country` column. Note that two of the values hashed to the same value (how convenient for the example!). While you won't see the internals, there are DMVs that will tell you the average bucket length, and it will almost never be 1, even when the index key is unique.

WOW! eBook
www.wowebook.org

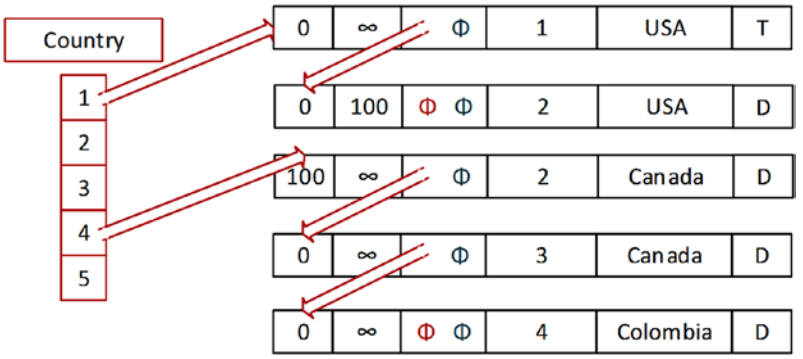


Figure 10-20. *Sample hash index*

In the data, USA hashes to bucket 1 (coincidentally, of course), and Canada and Colombia hash to 5. When SQL Server does a lookup in the index, it uses the same hash function that it used to put up the value, then it goes to the first row in the hash bucket, and scans all of the rows. So it would be three memory reads for Colombia, or Canada, or any other value that hashes to 4.

Indexing In-Memory OLTP Tables

Microsoft's current prescription as of SQL Server 2016 for indexing in-memory OLTP tables is to start with the Bw-Tree based nonclustered indexes for pretty much all primary keys, unique keys, and all other indexes one might need. They are easier to create and have less maintenance, and they work well for most of the use cases one will have for in-memory OLTP tables.

It must be noted, however, that hash indexes will perform better as long as your use case is 100% single-value lookups like a primary key. This is largely due to fixed memory sizing and less processing needed to get to the head of the linked list of rows. But even then, how well the hashing function performs will depend on how many rows will end up needing scanned, as opposed to an NC Bw-Tree, where the scanned rows will be based strictly on all rows having the same index key value. Be certain that your bucket sizes are at least twice the number of unique values you have/expect and watch the chain length as previously mentioned. A very large number of values in a chain will start to degrade performance. Also remember that both indexes will have in their index chains data for expired rows that have not been cleaned up.

In the `WideWorldImporters` sample database, there are a few in-memory tables. We will use `Warehouse.VehicleTemperatures` in this section for a few plan demonstrations. It has the following structure:

```
CREATE TABLE Warehouse.VehicleTemperatures
(
        VehicleTemperatureID bigint IDENTITY(1,1) NOT NULL
                CONSTRAINT PK_Warehouse_VehicleTemperatures  PRIMARY KEY NONCLUSTERED,
        VehicleRegistration nvarchar(20) COLLATE Latin1_General_CI_AS NOT NULL,
        ChillerSensorNumber int NOT NULL,
        RecordedWhen datetime2(7) NOT NULL,
        Temperature decimal(10, 2) NOT NULL,
        FullSensorData nvarchar(1000) COLLATE Latin1_General_CI_AS NULL,
        IsCompressed bit NOT NULL,
        CompressedSensorData varbinary(max) NULL
) WITH ( MEMORY_OPTIMIZED = ON , DURABILITY = SCHEMA_AND_DATA );
```


WOW! eBook
www.wowebook.org

It comes with 65,998 rows, and does not have any other indexes or foreign keys defined on the table itself other than the primary key.

The first thing you will notice is that not much is different in using these tables in simple queries (differences show up more in the next chapter with explicit transactions and concurrency, and even more when we get to Chapter 13 and I discuss a bit about native code versus interpreted T-SQL). Take the following query:

```
SELECT *
FROM   Warehouse.VehicleTemperatures;
```

The query plan is very much what you would expect:

```
|--Table Scan(OBJECT:([WideWorldImporters].[Warehouse].[VehicleTemperatures]))
```

Add in a primary key value:

```
SELECT *
FROM   Warehouse.VehicleTemperatures
WHERE  VehicleTemperatureID = 2332;
```

and you get an index seek and a parameterized plan:

```
|--Index Seek (OBJECT:([WideWorldImporters].[Warehouse].[VehicleTemperatures].
                                                    [PK_Warehouse_VehicleTemperatures]),
               SEEK:([WideWorldImporters].[Warehouse].[VehicleTemperatures].
               [VehicleTemperatureID]=CONVERT_IMPLICIT(bigint,[@1],0)) ORDERED FORWARD)
```

One difference you may see is a plan making more use of an index. For example, the query

```
SELECT *
FROM   Warehouse.VehicleTemperatures
WHERE  VehicleTemperatureID <> 0;
```

looks like a definite case for a full table scan. `VehicleTemperature = 0` could only return 1 row, but the plan here is

```
|--Index Seek (OBJECT:([WideWorldImporters].[Warehouse].[VehicleTemperatures].
                                                    [PK_Warehouse_VehicleTemperatures]),
               SEEK:(
        [WideWorldImporters].[Warehouse].[VehicleTemperatures].[VehicleTemperatureID] < (0)
     OR [WideWorldImporters].[Warehouse].[VehicleTemperatures].[VehicleTemperatureID] > (0))
                                                                    ORDERED FORWARD)
```


WOW! eBook
www.wowebook.org

In testing this query with or without an index, using STATISTIC TIME:

```
SELECT *
FROM   Warehouse.VehicleTemperatures WITH (INDEX = 0)
WHERE  VehicleTemperatureID <> 0;
```

the time difference was consistently slower using the index, but generally less than .1 second. The point of in-memory OLTP tables is to do small transactions, so they expect you to know what you are doing.

One of the reasons that the prescription is to use Bw-Tree indexes first is that hash indexes have only one purpose: single-row lookups. So if you need to do any sort of range query, hash indexes will not be helpful. For example, let's add one to the RecordedWhen column:

```
ALTER TABLE  Warehouse.VehicleTemperatures
 ADD INDEX RecordedWhen                              --33000 distinct values,
    HASH (RecordedWhen) WITH (BUCKET_COUNT = 64000) --values are in powers of 2
```

Use an equality operator:

```
SELECT *
FROM   Warehouse.VehicleTemperatures
WHERE  RecordedWhen = '2016-03-10 12:50:22.0000000';
```

and the index is used:

```
|--Index Seek(
             OBJECT:([WideWorldImporters].[Warehouse].[VehicleTemperatures].[RecordedWhen]),
             SEEK:([WideWorldImporters].[Warehouse].[VehicleTemperatures].[RecordedWhen]
                                  =CONVERT_IMPLICIT(datetime2(7),[@1],0)) ORDERED FORWARD)
```

But even if the plan clearly can only return the same row, if it isn't equality, it will use a scan and filter:

```
SELECT *
FROM   Warehouse.VehicleTemperatures
WHERE  RecordedWhen BETWEEN '2016-03-10 12:50:22.0000000' AND '2016-03-10 12:50:22.0000000';
```

This results in the following plan:

```
|--Filter(
           WHERE:([WideWorldImporters].[Warehouse].[VehicleTemperatures].[RecordedWhen]>=
                                                 CONVERT_IMPLICIT(datetime2(7),[@1],0)
                  AND
                  [WideWorldImporters].[Warehouse].[VehicleTemperatures].[RecordedWhen]<=
                                                 CONVERT_IMPLICIT(datetime2(7),[@2],0)))
       |--Table Scan(OBJECT:([WideWorldImporters].[Warehouse].[VehicleTemperatures]))
                                                                          ORDERED FORWARD)
```

If you have throughput needs that merit the use of the in-memory engine, you need to do testing to make sure that everything is working as you expect. In Chapter 11, we will see how even fast-looking queries can have a negative impact on concurrency when you don't get your indexes right. In-memory OLTP is very new, and built for specific scenarios (hence "OLTP" in the title).

WOW! eBook
www.wowebook.org

Columnstore Indexes

Columnstore indexes are, for the reporting specialist, an amazing tool. In this section I just want to give a very brief introduction to the structures so you can see conceptually the difference between columnstore indexes and typical row store indexes. A columnstore index is a form of columnar index. Columnar databases have been around for quite a while, which store all data by column. They have generally been great at implementing reporting solutions where you have to scan a lot of rows for every query, especially when dealing with lots of duplicate data. They are particularly useful when processing aggregates on large sets. The more columns in the object, and the fewer you need to use in the query, the greater the performance benefit. Figure 10-21 shows a conceptual diagram of the structure of a columnstore index.

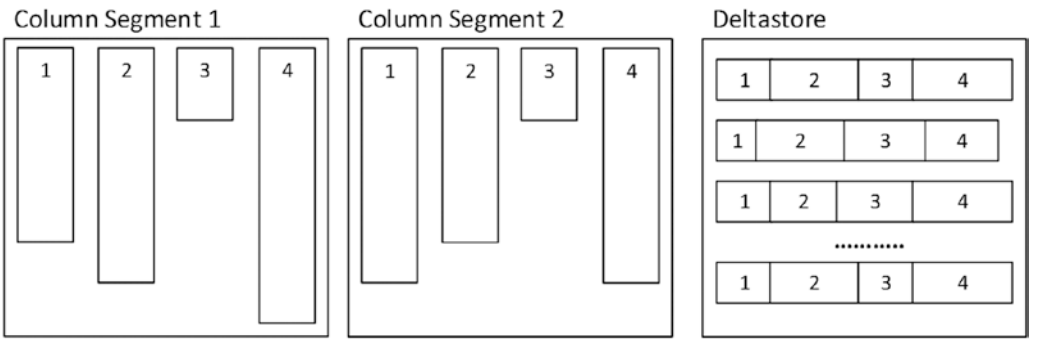


Figure 10-21. *Conceptual view of a columnstore index*

The Column Segment blocks are ideally 1,048,576 rows each with each of the numbered sections representing a column, with each segment ordered by the rows in the table. They are different sized because each of the segments is compressed, much like you can compress normal data on a page, but instead of an 8K page, it is compressed over 1 million rows in a blob style storage.

Data in a columnstore index is not ordered, but is scanned for all usage in what is called "batch mode," which processes data in ~900 at once, instead of the row-by-row mode used for typical row store indexes. In practice I have found in extreme cases (like a data warehouse fact table that has 120 columns and you are aggregating on 3) query performance can be orders of magnitude quicker.

The deltastore comes into play when you are modifying a table with a clustered columnstore index on it, new rows are dropped into the deltastore until you reach the 1,048,576-row threshold. The deltastore is basically a heap structure that will be scanned. Using the `REBUILD` or `REORGANIZE` commands of the `ALTER INDEX` command, you can manually push rows into the column segments also. I won't dig too deep into the management of columnstore indexes in this book.

Updates to rows in a columnstore index are a delete from the columnstore index, and then the row is added to either the deltastore for a clustered columnstore, or to the last columnstore segment for a nonclustered version. So it will behoove you to avoid doing a lot of modifications if possible.

In SQL Server 2016, there are clustered and nonclustered columnstore indexes, and both are updateable, and both can be mixed with row store indexes. This improvement for 2016 enables some very advanced scenarios, which will be covered in Chapter 14. The most important benefit of mixing the indexes is that columnstore indexes are horrible at single-row lookups. This is because there is no uniqueness declared or enforced. This leads to every query being a scan of the structure.

WOW! eBook
www.wowebook.org

Common OLTP Patterns of Index Usage

In this section I have gathered a few topics that are important to cover but did not fit the flow of the previous sections (sometimes because they needed multiple sections to be completed).

We will look at these topics:

- *When to cluster on something other than the primary key*: It may be the most common usage of the clustered index, but it isn't always best.
- *Indexing foreign keys*: Foreign keys almost always represent a common path to retrieve data from a table, but do they always need indexed? Do they ever? (Yes they do, sometimes!)
- *Indexed views*: So far we have limited discussion to indexing tables, but you can index views in some cases to give you fast answers maintained by SQL Server's code instead of yours.
- *Compression*: Fitting more data onto disk means less reads from disk, which is still the slowest part of a computer.
- *Partitioning*: Partitioning breaks up a table or index physically but still makes it look like one object to the end user. This allows for some coding and maintenance scenarios to be easier.

When to Cluster on Something Other Than the PRIMARY KEY

For an on-disk table, most every usage of a clustered index will be for the primary key, particularly when you are using a surrogate key. However, 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 for which 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*: Since the data is stored with the clustered index, you can avoid the cost of the bookmark lookup, which often results in a table scan in any case.
- *Child table foreign key references*: In some cases, a table like `InvoiceLineItem` with basic structure (`InvoiceLineItemId PK, InvoiceId FK, LineNumber, UNIQUE(InvoiceId, LineNumber)`) may benefit greatly by clustering on the `InvoiceId`, even though it would not be unique. First you fetch the `Invoice`, then the `InvoiceLineItem` rows. Important to note is that sometimes you will end up using the surrogate key to fetch rows more than seems logical, so empirical testing and watching usage are important.

The choice of how to pick the clustered index depends on a couple of 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

WOW! eBook
www.wowebook.org

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 start with a surrogate key for my clustering key, usually one of the integer types or sometimes even the unique identifier (GUID) type (ideally the sequential type), even if that is 16 bytes wide. I use the surrogate key because so many of the queries you do for modification (the general goal of the OLTP system) will typically 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.

To reiterate an important point, it is also very good to use the clustered index on a value like a monotonically increasing surrogate so that page splits over the entire index are greatly decreased on inserts. 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`. We will discuss this more in the "Indexing Dynamic Management View Queries" section later in the chapter.

Indexing Foreign Keys

Foreign key columns are a special case where we often need an index of some type. This is true both in on-disk and in-memory tables. 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 may not 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. Often when you have a parent table with specific values, you may 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. If in doubt, it is not a terrible practice to add indexes to foreign keys by default, and during testing use one of the queries in the upcoming "Indexing Dynamic Management View Queries" section to identify how or if indexes are being used.

Say you have five values in the parent table and 500 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 10-22.

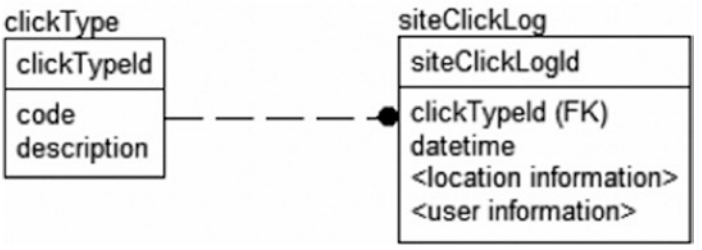


Figure 10-22. *Sample foreign key relationship*

WOW! eBook
www.wowebook.org

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 matching value in the table, if you don't have an index on the foreign key in the siteClickLog table, it may 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 500 million rows for the value. From the statistics on the column, the query processor 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 asynchronously. (Ironically, because it is an existence search, the query processor could fail quickly 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.

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   deleted;
```

Then, 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 relationships*: Used to implement a multivalued attribute of the parent
- *Many-to-many resolution table relationships*: 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

WOW! eBook
www.wowebook.org

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.

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 10-23.

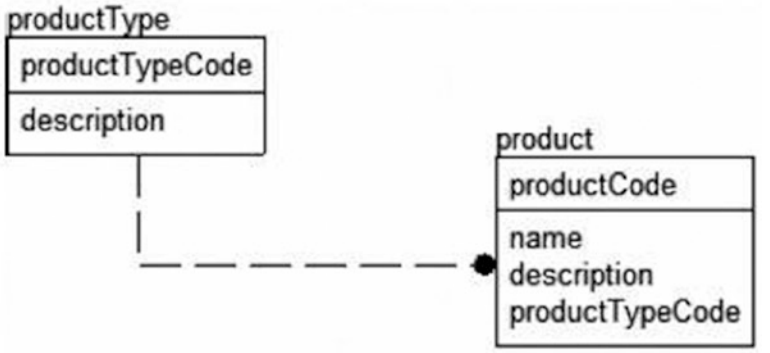


Figure 10-23. *Sample domain table relationship*

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 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 earlier in the chapter, 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 10-24.

WOW! eBook
www.wowebook.org

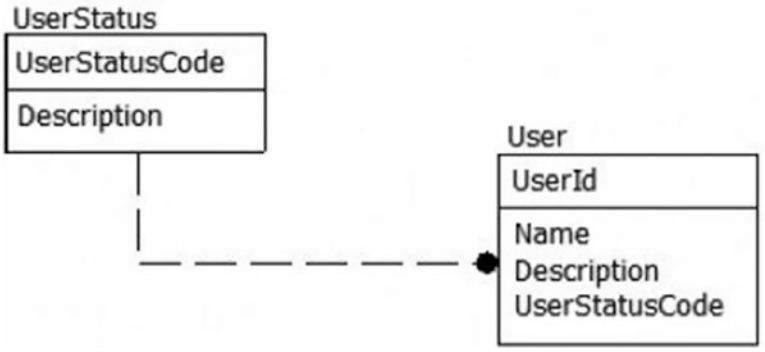


Figure 10-24. *Sample domain table relationship with low cardinality*

In this case, most users would be in the database with an active status. However, when a user is 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 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 10-25.

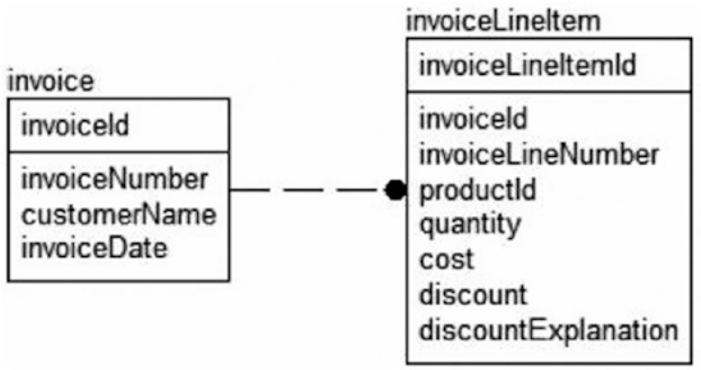


Figure 10-25. *Sample ownership relationship*

In this case, it's essential to have an index on the `invoiceLineItem.invoiceId` column, probably as part of a `UNIQUE` constraint. A large amount of 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).

WOW! eBook
www.wowebook.org

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, as I noted in the previous section about when to cluster on a non-surrogate value. 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.

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. Using an example that we used in Chapter 7, with a many-to-many relationship between tables that hold games a person owns, the relationship between `gamePlatform` and `game` is shown in Figure 10-26.

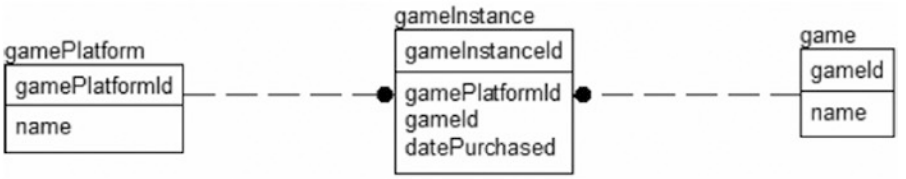


Figure 10-26. *Sample many-to-many relationship*

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`).

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 10-27.

WOW! eBook
www.wowebook.org

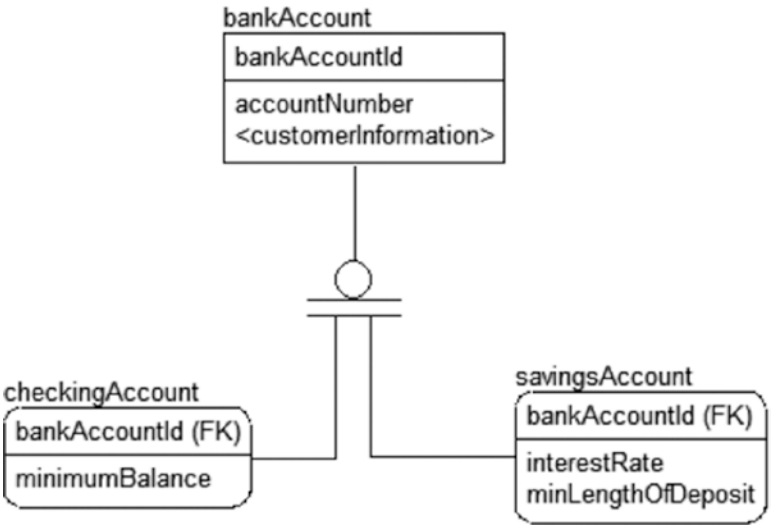


Figure 10-27. *Sample one-to-one relationship*

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.

Indexed 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 (more or less) 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 Enterprise edition or greater (plus 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 `WideWorldImporters` database. Note that only schema-bound views can be indexed as this makes certain that the tables and structures that the index is created upon won't change underneath the view. (A more complete list of requirements is presented after the example.)

```
CREATE VIEW Warehouse.StockItemSalesTotals
WITH SCHEMABINDING
AS
SELECT StockItems.StockItemName,
```


WOW! eBook
www.wowebook.org

```
                              --ISNULL because expression can't be nullable
       SUM(OrderLines.Quantity * ISNULL(OrderLines.UnitPrice,0)) AS TotalSalesAmount,
       COUNT_BIG(*) AS TotalSalesCount--must use COUNT_BIG for indexed view
FROM  Warehouse.StockItems
          JOIN Sales.OrderLines
                 ON  OrderLines.StockItemID = StockItems.StockItemID
GROUP  BY StockItems.StockItemName;
```

This would do the calculations at execution time. We run the following query:

```
SELECT *
FROM   Warehouse.StockItemSalesTotals;
```

The plan looks like…well, like five operators that go on for a while and would be really hard to read in text. An image is provided in Figure 10-28. It loses a lot of fidelity from the textual plans, but it might save a tree.

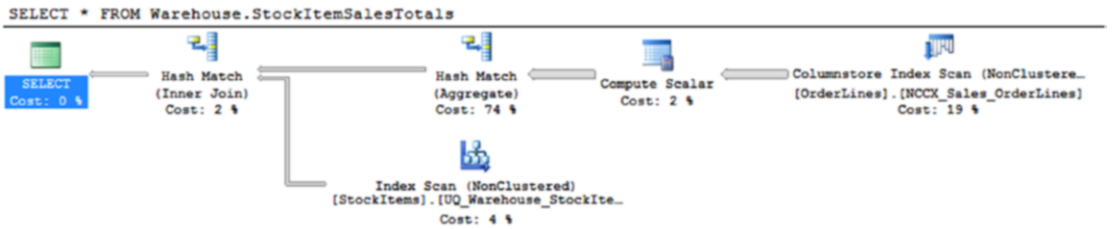


Figure 10-28. *Query plan image from using the view*

This is a big plan for such a small query, for sure, and it's hard to follow. It scans the `OrderLines` 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 only returns 227 rows in this sample database, but say this query wasn't fast enough, or it used too many resources to execute, or it was 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 XPKStockItemSalesTotals on
                     Warehouse.StockItemSalesTotals(StockItemName);
```

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 and the plan looks like the following:

```
|--Clustered Index Scan
             (OBJECT:([WideWorldImporters].[Warehouse].[StockItemSalesTotals].
                                                          [XPKStockItemSalesTotals]))
```


WOW! eBook
www.wowebook.org

No big deal, right? We expected this result because we directly queried the view. On my test system, running Developer edition (which is functionally comparable to Enterprise edition), you get a great insight into how cool this feature is by running the following query, which is basically our original query with a division operator between the two factors:

```
SELECT StockItems.StockItemName,
       SUM(OrderLines.Quantity * ISNULL(OrderLines.UnitPrice,0)) / COUNT_BIG(*)
                                                                  AS AverageSaleAmount
FROM  Warehouse.StockItems
          JOIN Sales.OrderLines
                ON  OrderLines.StockItemID = StockItems.StockItemID
GROUP  BY StockItems.StockItemName;
```

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]=
               [WideWorldImporters].[Warehouse].[StockItemSalesTotals].[TotalSalesAmount]
    /CONVERT_IMPLICIT(decimal(19,0),[WideWorldImporters].[Warehouse].[StockItemSalesTotals].
                                                                    [TotalSalesCount],0)))
       |--Clustered Index Scan
              (OBJECT:([WideWorldImporters].[Warehouse].[StockItemSalesTotals].
                                                                  [XPKStockItemSalesTotals]))
```

There is a scalar compute that does our math, but instead you will notice that the plan references the `StockItemSalesTotals` 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 give them performance they didn't even ask for.

■ **Tip** The indexed view feature in 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.

WOW! eBook
www.wowebook.org

- 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 "Create Indexed Views" (`msdn.microsoft.com/en-us/library/ms191432.aspx`). 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.

■ **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, in any case where you can handle some latency on the data

---

Compression

I mentioned compression when we looked at columnstore indexes. In a normal index (which includes clustered, heaps, and B-Trees), SQL Server can save space by storing all of your data like it was a variable-sized type, yet in usage, the data will appear and behave like a fixed length type. In Appendix A, I will note how compression will affect each datatype individually, or if you want to see a list that may show any recent changes, you can check SQL Server Books Online for the topic of "Row Compression Implementation" (`msdn.microsoft.com/en-us/library/cc280576.aspx`).

For example, if you stored the value of 100 in an `int` column, SQL Server needn't use all 4 bytes; it can store the value 100 in the same amount of space as a `tinyint`. So instead of taking a full 4 bytes, SQL Server can simply use 8 bits (1 byte). Another case is when you use a fixed length type like a `char(30)` column but store only two characters; 28 characters could be saved, and the data padded as it is used.

This datatype-level compression is part of *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. Row compression is a very interesting option 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:

WOW! eBook
www.wowebook.org

- *Prefix compression*: The storage engine 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.

```
USE Tempdb
GO
CREATE TABLE dbo.TestCompression
(
    TestCompressionId int,
    Value  int
)
WITH (DATA_COMPRESSION = ROW) -- PAGE or NONE
    ALTER TABLE testCompression REBUILD WITH (DATA_COMPRESSION = PAGE);

CREATE CLUSTERED INDEX Value
   ON testCompression (Value) WITH ( DATA_COMPRESSION = ROW );

ALTER INDEX Value  ON testCompression REBUILD WITH ( DATA_COMPRESSION = PAGE );
```

■ **Note** The syntaxes of the `CREATE INDEX` and `CREATE TABLE` commands allow 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 it in your production systems.

WOW! eBook
www.wowebook.org

Partitioning

Partitioning allows you to break an index/table into multiple physical structures that look to the end user to be only one by breaking them into more manageable chunks. Partitioning can allow SQL Server to scan data from different processes, enhancing opportunities for parallelism.

Partitioning allows you to define physical divisions within your 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.

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 tune your databases, particularly if they are very large or very active. In-memory tables can participate in a partitioning scheme, and not every partition needs to be in-memory to do so. For deeper coverage, I would suggest you consider one 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. 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 our example:

```
USE Tempdb;
GO
--Note that the PARTITION FUNCTION is not a schema owned object
CREATE PARTITION FUNCTION PartitionFunction$Dates (date)
AS RANGE LEFT FOR VALUES ('20140101','20150101');
                  --set based on recent version of
                  --WideWorldImporters.Sales.Orders 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 <= '20131231'`
- `value >= '20140101'` and `value <= '20141231'`
- `value >= '20150101'`

Next, use that partition function to create a partitioning scheme:

```
CREATE PARTITION SCHEME PartitonScheme$dates
                AS PARTITION PartitionFunction$dates ALL to ( [PRIMARY] );
```

This 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'.
```


WOW! eBook
www.wowebook.org

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 may 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.

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.Orders
(
    OrderId     int,
    CustomerId  int,
    OrderDate  date,
    CONSTRAINT PKOrder PRIMARY KEY NONCLUSTERED (OrderId) ON [Primary],
    CONSTRAINT AKOrder UNIQUE CLUSTERED (OrderId, OrderDate)
) ON PartitonScheme$dates (OrderDate);
```

Next, load some data from the WideWorldImporters.Sales.Orders table to make looking at the metadata more interesting. You can do that using an INSERT statement such as the following:

```
INSERT INTO dbo.Orders (OrderId, CustomerId, OrderDate)
SELECT OrderId, CustomerId, OrderDate
FROM  WideWorldImporters.Sales.Orders;
```

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 partition
FROM   dbo.Orders;
```

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.Orders');
```

This will return the following:

```
partition_number index_id    rows        name           type_desc
---------------- ----------- ----------- -------------- ---------------------
1                1           19547       AKSalesOrder   CLUSTERED
2                1           21198       AKSalesOrder   CLUSTERED
3                1           32850       AKSalesOrder   CLUSTERED
1                2           73595       PKSalesOrder   NONCLUSTERED
```


WOW! eBook
www.wowebook.org

Partitioning is not a general-purpose tool that should be used on every table. 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). Multiple uniqueness constraints can be difficult using rolling windows, so be sure to cover all of your uniqueness needs in some manner.
- *Maintenance*: Some maintenance can be done at the partition level rather than for the entire table, so once partition data is read-only, you may not need to maintain any longer. Some caveats do apply that are subject to change, so I will leave it to you to check the documentation.

Indexing 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 in 2010, is *my* favorite book on the subject, even though it is kind of old at this point). 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 provides a peek at what indexes the optimizer thought might be useful for queries (on-disk and in-memory). It can be very helpful when tuning a database, particularly a very busy database executing thousands of queries a minute. I have personally used this query to tune third-party systems where I didn't have a lot of access to the queries in the system and using Extended Events was simply far too cumbersome with too many queries to effectively tune manually.

This query uses three of the dynamic management views that are part of the missing index family of objects:

- `sys.dm_db_missing_index_groups`: This relates missing index groups with the indexes in the group.
- `sys.dm_db_missing_index_group_stats`: This provides statistics of how much the indexes in the group 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.

WOW! eBook
www.wowebook.org

The query is as follows. I won't attempt to build a scenario where you can test this functionality 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
         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. Ignore these for in-memory queries as all columns are essentially included due to the structure of these tables.

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 that 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

WOW! eBook
www.wowebook.org

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 article I read about using the missing indexes, "Fun for the Day – Automated Auto-Indexing" (blogs.msdn.microsoft.com/queryoptteam/2006/06/01/fun-for-the-day-automated-auto-indexing/). 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:

```
object_name                          equality_columns              inequality_columns
------------------------------------ ----------------------------- -------------------
databasename.schemaname.tablename    columnfirst, columnsecond     columnthird

included_columns
------------------------
columnfourth, columnfifith
```

You could build the following index to satisfy the need:

```
CREATE INDEX XName ON databaseName.schemaName.TableName(columnfirst, columnsecond,
columnthird) INCLUDE (columnfourth, columnfifith);
```

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.

WOW! eBook
www.wowebook.org

However, limitations aside, the missing index dynamic management views 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.

On-Disk 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 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.

WOW! eBook
www.wowebook.org

- `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.

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 i
             ON f.[object_id] = i.[object_id] AND f.[index_id] = i.[index_id]
        JOIN sys.objects o
             ON i.[object_id] = o.[object_id]
        JOIN sys.schemas 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 `UNIQUE` constraint `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.

WOW! eBook
www.wowebook.org

In-Memory OLTP Index Stats

In this section I want to briefly point out a few DMVs that you can use with in-memory tables to get some general-purpose information about your objects.

The first is `sys.dm_db_xtp_table_memory_stats`, and it will provide you information about how much memory is being tied up by your in-memory objects (note the `xtp` as part of the names; an early naming convention was extreme programming, and it ended up in the name):

```
SELECT OBJECT_SCHEMA_NAME(object_id) + '.' +
       OBJECT_NAME(object_id) AS objectName,
           memory_allocated_for_table_kb,memory_used_by_table_kb,
           memory_allocated_for_indexes_kb,memory_used_by_indexes_kb
FROM sys.dm_db_xtp_table_memory_stats;
```

In the results, you can see in kilobytes how much memory is allocated to your objects, and how much of that allocation is actually used from the tables, and by the indexes. The next object is sys.dm_db_xtp_index_stats, and here is a basic query to use:

```
SELECT OBJECT_SCHEMA_NAME(ddxis.object_id) + '.' +
       OBJECT_NAME(ddxis.object_id) AS objectName,
           ISNULL(indexes.name,'BaseTable') AS indexName,
           scans_started, rows_returned, rows_touched,
           rows_expiring, rows_expired,
           rows_expired_removed, phantom_scans_started --and several other phantom columns
FROM   sys.dm_db_xtp_index_stats AS ddxis
                JOIN sys.indexes
                       ON indexes.index_id = ddxis.index_id
                         AND indexes.object_id = ddxis.object_id;
```

This gives us a couple of interesting things. The number of times the index was used is in scans_started, the number of rows returned and touched in queries are all documented objects. There are other internal columns listed that will show you some details about expiring rows, and phantom scans. We will talk about phantom rows in the next chapter, but due to the way in-memory tables implement concurrency, if you are in `SERIALIZABLE` isolation level, a scan must be done at commit time to ensure nothing has changed.

Finally, if you chose to use hash indexes, you will want to use `sys.dm_db_xtp_hash_index_stats` to check up on its structures:

```
SELECT OBJECT_SCHEMA_NAME(ddxhis.object_id) + '.' +
       OBJECT_NAME(ddxhis.object_id) AS objectName,
           ISNULL(indexes.name,'BaseTable') AS indexName,
           ddxhis.total_bucket_count, ddxhis.empty_bucket_count,
           ddxhis.avg_chain_length, ddxhis.max_chain_length
FROM   sys.dm_db_xtp_hash_index_stats ddxhis
                JOIN sys.indexes
                       ON indexes.index_id = ddxhis.index_id
                         AND indexes.object_id = ddxhis.object_id;
```


WOW! eBook
www.wowebook.org

This will return to you the bucket count as it was actually created (bucket counts are implemented in powers of 2, the number of empty buckets, average chain length (or number of rows that have pointers to each other), and max chain length. Using the `RecordedWhen` index on `Warehouse.VehicleTemperatures` we created earlier in the chapter:

```
total_bucket_count   empty_bucket_count   avg_chain_length     max_chain_length
-------------------- -------------------- -------------------- --------------------
65536                39594                2                    10
```

We created 64,000 buckets, and it rounded up to 65,536. Of the 65,998 rows, there were 32,999 distinct values. An average chain length of 2 is good, and 10 rows max is typical in the cases I have seen, even with uniqueness constraint indexes.

Best Practices

Indexing is a complex subject, and even though this is not a short chapter, we've only scratched the surface. Add in the newer memory-optimized technologies, and our choices are getting numerous. The following best practices are what I use as a rule of thumb when creating a new database solution. I assume in these that you've applied `UNIQUE` constraints in all places where you have defined a uniqueness need. 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 trade-off. The first practice is the most important.

- *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, but in that case you should test to see if they are needed.
- *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*: For all indexes of any types, 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. Columnstore indexes can withstand much wider column counts, but if you will not use a column in a query, maybe not there either.
- *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.

WOW! eBook
www.wowebook.org

- `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.
  - 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.

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 need to see. In this chapter, we left the world of relational programming and took a peek 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 18 plus years (just counting from the major rewrite of SQL Server in version 7.0; and realizing that Microsoft just added the in-memory engine 4 years ago as I write this). Much of the T-SQL code written for version 1.0 would STILL WORK TODAY. Take that all other programmers. Most of it, with a little bit of translation, would even run using the new in-memory engine (if perhaps not taking full advantage, more of which I will cover in Chapter 13).

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.

WOW! eBook
www.wowebook.org

We looked at the physical structure of how SQL Server persists 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. When using in-memory, data is housed in memory, but backed up with—you guessed it—files.

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.

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.

WOW! eBook
www.wowebook.org

CHAPTER 11

Matters of Concurrency

If you try to multitask in the classic sense of doing two things at once, what you end up doing is quasi-tasking. It's like being with children. You have to give it your full attention for however much time you have, and then you have to give something else your full attention.

—Joss Whedon, American screenwriter, film and television director, and comic book author

Concurrency is the ability to have multiple users (or processes/requests) access (change) shared data at the same time. The key 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. Fortunately, SQL Server's execution engine can give its full attention to a task as long as it needs to, even if that is just microseconds (unlike we humans, as I've had to correct several mistakes I made as a result of trying to write this sentence while talking to someone else).

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 methods*: I'll explain the two major types of concurrency controls used in SQL Server—pessimistic (using locks) and optimistic (using row versions).
- *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.

Like the previous chapter, we will look at the differences in how the on-disk technologies contrast with the newer in-memory OLTP technologies. One of the largest differences between the two (once you get past some of the in-memory limitations) is how each handles concurrency.

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_11

WOW! eBook
www.wowebook.org

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. A secondary goal is to help you understand what you need to design into your code and structures to deal with the aftermath of SQL Server's coordination.

RESOURCE GOVERNOR

SQL Server has a feature called Resource Governor that is concurrency related (especially as it relates to performance tuning), though it is more of a management tool than a design concern. Resource Governor allows you to partition the workload of the entire server by specifying maximum and minimum resource allocations (memory, CPU, concurrent requests, IO, 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.).

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.

One nice thing about Resource Governor is that some settings 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.

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 trade-offs:

- *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? (This probably sounds horrible, but some concurrency tuning techniques lead to results that are actually wrong.)
- *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.

WOW! eBook
www.wowebook.org

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 actual work might be done, but just like the old days, people would sit around waiting for their turn (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).

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 as would be expected.

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.

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? If your database system is used to schedule vehicles like aircraft or trains, it is even less unacceptable.

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 performed 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 shared SQL Server resources. 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 trade-offs between performance, consistency, and the number of simultaneous users.

WOW! eBook
www.wowebook.org

■ **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.

OS and Hardware Concerns

SQL Server is designed to run on a variety of hardware types, from a simple laptop with one processor to massive machines with at processors, and once Windows Server 2016 arrives, it could be far larger. What is amazing is that essentially the same basic code runs on a low-end computer as well as 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 up to 1GB of RAM and one processor socket (with up to four cores, which is still more than our first SQL Server machine had one 486 processor and 16MB of RAM), and at the other end of the spectrum, Enterprise edition can handle as much hardware as a manufacturer can stuff into one box. Additionally, a specialized configuration called Parallel Data Warehouse is built specifically for data warehousing loads, and is the basis for the Azure SQL Data Warehouse product. Generally speaking, though, in every version, the very same concerns exist about how SQL Server handles multiple users using the same resources seemingly simultaneously. 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. My goal in this chapter isn't to delve too deeply into the gory hardware details, but it's important to mention that concurrency is heavily tied to hardware architecture. For example, consider the following subsystems:

- *Processor*: It controls the rest of the computer subsystems, as well as doing any calculations needed. If you have too few processors, less work can be done simultaneously, and excessive time can be wasted switching between requests.
- *Disk subsystem*: Disk is always the slowest part of the system (even with solid-state drives [SSDs] becoming almost the status quo). A slow disk subsystem is the downfall of many systems, particularly because of the expense involved. Each drive can only read 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. I won't go any deeper into disk configuration because the tech changes faster than you can imagine, but even SSDs won't solve every IO issue.

WOW! eBook
www.wowebook.org

- *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). Using `SET NOCOUNT ON` 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.
- *Memory*: One of the cheapest commodities that you can improve substantially on a computer is memory. SQL Server can use a tremendous amount of memory within the limits of the edition you use (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 128 CPUs and 1TB 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 drives, disk controllers, and network cards and the amount of RAM you have all affect concurrency.

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 (`msdn.microsoft.com`) or great blogs like Glenn Berry's (`sqlserverperformance.wordpress.com`). I'll be focusing on design- and coding-related issues pertaining to how to write better SQL code 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.

In this section we will first discuss a few details about transactions, followed by an overview of the syntax involved with base transactions.

Transaction Overview

Whenever data is modified in the database, the changes are not written to the physical table structures directly, 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 (though you can change the log write to be asynchronous as well with the `DELAYED DURABILITY` database setting). Later, the physical disk structure is written asynchronously. Understanding the process of how modifications to data are made is essential, because while tuning your overall system, you have to be cognizant that when every modification operation is logged, you need to consider how large to make your transaction log, and when a database is written to frequently, the data files are often less important than the log files.

Beyond being a container for modification operations, transactions provide a container 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, as well as defining boundaries for making sure one transaction's actions aren't affected by another more than is expected.

WOW! eBook
www.wowebook.org

To explain the purpose of transactions, there's a common acronym: ACID. It stands for the following:

- *Atomicity*: Every operation within a transaction appears to be 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, and physical data written is as expected.
- *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 done by several methods, covered in the "SQL Server Concurrency Methods" section later in this chapter.
- *Durability*: Once a transaction is completed (committed), all changes must be persisted if desired. The modifications should persist even in the event of a system failure. (Note that in addition to delayed durability, in-memory tables allow non-durable tables, which are empty on a server restart).

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 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. 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.

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.

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 (manually or automatically by SQL Server). 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 Books Online.

WOW! eBook
www.wowebook.org

Transaction Syntax

The syntax to start and stop transactions is pretty simple. I'll cover four variants of the transaction syntax in this section:

- *Transaction basics*: 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.

Transaction Basics

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 (particularly since you don't realize it is still open). 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` may have to wait (more on why this is throughout this chapter). The worst case of this I saw, a

WOW! eBook
www.wowebook.org

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 (the connection was a pooled connection from a web site, so it was not a happy solution to management either).

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;
```

The error message is as follows:

```
Msg 6401, Level 16, State 1, Line 7
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 (which you can tell by using `SELECT @@TRANCOUNT;`—if that does not return 0, there is an open transaction). 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 putting a mark in the transaction log, which can be used when restoring a transaction log, instead of trying to figure out the date and time when an operation occurred. 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.

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 `WideWorldImporters` database (which replaced `AdventureWorks` as the de facto standard example database for SQL Server 2016), I'll demonstrate this capability.

We first set up the scenario by putting the `WideWorldImporters` database in full recovery model. The version I downloaded came with the `SIMPLE` recovery model.

```
USE Master;
GO

ALTER DATABASE WideWorldImporters
      SET RECOVERY FULL;
```


WOW! eBook
www.wowebook.org

Next, we create a couple of backup devices to hold the backups we're going to do:

```
EXEC sp_addumpdevice 'disk', 'TestWideWorldImporters ',
                             'C:\temp\WideWorldImporters.bak';
EXEC sp_addumpdevice 'disk', 'TestWideWorldImportersLog',
                             'C:\temp\WideWorldImportersLog.bak';
```

■ **Tip** You can see the current setting using the following code:

```
SELECT  recovery_model_desc
FROM    sys.databases
WHERE   name = 'WideWorldImporters';
```

If you need to delete the dump device for some reason, use

```
EXEC sys.sp_dropdevice @logicalname = '<name>';
```

Next, we back up the database to the dump device we created:

```
BACKUP DATABASE WideWorldImporters TO TestWideWorldImporters;
```

Now, we change to the `WideWorldImporters` database and delete some data from a table:

```
USE WideWorldImporters;
GO
SELECT COUNT(*)
FROM   Sales.SpecialDeals;

BEGIN TRANSACTION Test WITH MARK 'Test';
DELETE Sales.SpecialDeals;
COMMIT TRANSACTION;
```

This returns `2` for the original amount of `SpecialDeals` rows. Run the `SELECT` statement again, and it will return `0`. Next, back up the transaction log to the other backup device:

```
BACKUP LOG WideWorldImporters TO TestWideWorldImportersLog;
```

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 WideWorldImporters FROM TestWideWorldImporters
                                            WITH REPLACE, NORECOVERY;

RESTORE LOG WideWorldImporters FROM TestWideWorldImportersLog
                                            WITH STOPBEFOREMARK = 'Test', RECOVERY;
```


WOW! eBook
www.wowebook.org

Now, execute the counting query again, and you can see that the 29 rows are in there:

```
USE WideWorldImporters;
GO
SELECT COUNT(*)
FROM   Sales.SpecialDeals;
```

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.

Nested Transactions

Every time I hear the term "nested" transactions, I envision Marlin Perkins in some exotic locale about to tell us the mating habits of transactions, but you already know that is not what this is about (and unless you are of a certain age, have no idea who that is anyhow). I am referring to starting a transaction after another transaction has already been started. You can nest the starting of transactions like the following, allowing code to call other code that also starts a transaction:

```
BEGIN TRANSACTION;
    BEGIN TRANSACTION;
       BEGIN TRANSACTION;
```

In the engine, 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, use the `@@TRANCOUNT` global variable as previously mentioned. 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. If you have an actual use case, I will buy you lunch at the PASS Summit or any SQL Saturday!)

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
```


WOW! eBook
www.wowebook.org

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 previous 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
```

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 ends 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;
```


WOW! eBook
www.wowebook.org

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, or anytime a transaction has not been started, 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.
```

Autonomous Transactions

The concept of an *autonomous transaction* is that a transaction can occur within another transaction and commit even if the external transaction does not. SQL Server does not have the ability to do a user-defined autonomous transaction, but there is one example of such a thing in SQL Server. Back in Chapter 6, I introduced the `SEQUENCE` object, as well as a column using the `IDENTITY` property. The transactions that these objects use operate are autonomous to the external transaction. So if you fetch a new `SEQUENCE` or `IDENTITY` value, but then `ROLLBACK` the external transaction, the value is lost.

As an example, consider the following `SEQUENCE` and `TABLE` objects (built in a database named `Chapter11` in the downloads…and I too cringe at the name `Chapter11`!):

```
CREATE SCHEMA Magic;
GO
CREATE SEQUENCE Magic.Trick_SEQUENCE AS int START WITH 1;
GO
CREATE TABLE Magic.Trick
(
        TrickId int NOT NULL IDENTITY,
        Value int CONSTRAINT DFLTTrick_Value DEFAULT (NEXT VALUE FOR Magic.Trick_SEQUENCE)
)
```


WOW! eBook
www.wowebook.org

Now every time you execute the following code, you will get one row back, with incrementing numbers. You could run it on hundreds of connections, and you would get back two numbers that are growing larger.

```
BEGIN TRANSACTION;
INSERT INTO Magic.Trick DEFAULT VALUES; --just use the default values from table
SELECT * FROM Magic.Trick;
ROLLBACK TRANSACTION;
```

And "poof!" the rows disappear due to the rollback. There would never be more than one row in the output, but the values change. There are two transactions occurring for the IDENTITY and SEQUENCE, and those are not rolled back. Only the INSERT operation. This is interesting because, as we will see later, when we look at isolation of connections, other connections will not need to wait to see if that value is used, because that autonomous transaction has been committed.

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: *savepoints*. Unfortunately, using savepoints requires forethought and a special syntax, which makes them slightly difficult to use without careful planning. Savepoints are used to provide "selective" rollback.

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, I will use the following table:

```
CREATE SCHEMA Arts;
GO
CREATE TABLE Arts.Performer
(
    PerformerId int IDENTITY CONSTRAINT PKPeformer PRIMARY KEY,
    Name varchar(100)
 );
```

Next I will insert two different performers, one I like, and one that should never have been included in my book:

```
BEGIN TRANSACTION;
INSERT INTO Arts.Performer(Name) VALUES ('Elvis Costello');

SAVE TRANSACTION savePoint; --the savepoint name is case sensitive, even if instance is not
                            --if you do the same savepoint twice, the rollback is to latest
```


WOW! eBook
www.wowebook.org

```
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 code 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 use 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 when the transaction is enlisted into a distributed transaction.

Distributed Transactions

It would be wrong not to at least bring up the subject of distributed transactions. Occasionally, you might need to view or update data on a server that's different from the one on which your code resides. The Microsoft Distributed Transaction Coordinator (MS DTC) service 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` for the current setting, and set the value using `sp_configure`). Note also that savepoints aren't supported for distributed transactions.

WOW! eBook
www.wowebook.org

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
```

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.

Transaction State

There are three states that a transaction can be in, which can be detected by the return value from the `XACT_STATE()` system function:

- `1` (Active Transaction): A transaction has been started. No indication of how many levels of transaction have been nested are provided, just that there is an active transaction. As mentioned earlier, use `@@TRANCOUNT` for nested transaction information.
- `0` (No Active Transactionerror handlers): The connection is not currently in the context of a transaction
- `-1` (Uncommittable Transaction): Also known as a "doomed" transaction, something has occurred that makes the transaction still active but no longer able to be committed.

WOW! eBook
www.wowebook.org

The first two states are just as we have discussed earlier, but the uncommittable transaction is the special case that is most commonly associated with error handling and XACT_ABORT or triggers. If you hit an error in a TRY CATCH block while using XACT_ABORT or a trigger returns an error without rolling back, you can end up in an uncommittable state. As an example, let's create the following table:

```
CREATE SCHEMA Menu;
GO
CREATE TABLE Menu.FoodItem
(
    FoodItemId int NOT NULL IDENTITY(1,1)
        CONSTRAINT PKFoodItem PRIMARY KEY,
    Name varchar(30) NOT NULL
        CONSTRAINT AKFoodItem_Name UNIQUE,
    Description varchar(60) NOT NULL,
        CONSTRAINT CHKFoodItem_Name CHECK (LEN(Name) > 0),
        CONSTRAINT CHKFoodItem_Description CHECK (LEN(Description) > 0)
);
```

The constraints make certain that the length of the string columns is > 0. Now create a trigger that prevents someone from inserting 'Yucky' food in our database. Clearly this is not a complete solution, but it will cause the trigger to not let the insert occur.

```
CREATE TRIGGER Menu.FoodItem$InsertTrigger
ON Menu.FoodItem
AFTER INSERT
AS --Note, minimalist code for demo. Chapter 7 and Appendix B
   --have more details on complete trigger writing
BEGIN
   BEGIN TRY
               IF EXISTS (SELECT *
                                    FROM Inserted
                                    WHERE Description LIKE '%Yucky%')
        THROW 50000, 'No ''yucky'' food desired here',1;
   END TRY
   BEGIN CATCH
       IF XACT_STATE() <> 0
          ROLLBACK TRANSACTION;
       THROW;
   END CATCH;
END
GO
```

In this initial version of the trigger, we do roll back the transaction. Later in the section we will remove the error handling from the trigger, as well as the ROLLBACK to see what occurs. To show how this works, we will use XACT_ABORT, which without error handling will stop the batch. It is a very useful tool when writing maintenance scripts.

WOW! eBook
www.wowebook.org

```
SET XACT_ABORT ON;

BEGIN TRY
    BEGIN TRANSACTION;

        --insert the row to be tested
        INSERT INTO Menu.FoodItem(Name, Description)
        VALUES ('Hot Chicken','Nashville specialty, super spicy');

        SELECT  XACT_STATE() AS [XACT_STATE], 'Success, commit'  AS Description;
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
        IF XACT_STATE() = -1 --transaction not doomed, but open
          BEGIN
                SELECT -1 AS [XACT_STATE], 'Doomed transaction'  AS Description;
                ROLLBACK TRANSACTION;
          END
        ELSE IF XACT_STATE() = 0 --transaction not doomed, but open
          BEGIN
                SELECT 0 AS [XACT_STATE], 'No Transaction'  AS Description;;
          END
        ELSE IF XACT_STATE() = 1 --transaction still active
          BEGIN
                SELECT 1 AS [XACT_STATE],
                       'Transction Still Active After Error'  AS Description;
                ROLLBACK TRANSACTION;
          END
END CATCH;
```

Since this insert met the requirements of all constraints, it succeeds:

```
XACT_STATE Description
---------- ---------------
1          Success, commit
```

Next, we will use the following insert (wrapped with the error handler from above) with an empty string for the description because, honestly, no words could adequately describe what that food probably would taste like:

```
INSERT INTO Menu.FoodItem(Name, Description)
VALUES ('Ethiopian Mexican Vegan Fusion','');
```

Because there was a constraint violated, the transaction is doomed, so we get the following:

```
XACT_STATE  Description
----------- ------------------
-1          Doomed transaction
```


WOW! eBook
www.wowebook.org

When the trigger does a ROLLBACK because of the string "yucky" being found

```
INSERT INTO Menu.FoodItem(Name, Description)
VALUES ('Vegan Cheese','Yucky imitation for the real thing');
```

we come out of the INSERT with no transaction:

```
XACT_STATE Description
---------- ---------------
0          No Transaction
```

Finally, the last scenario I will show is the case where a trigger just returns an error, and does not roll back:

```
ALTER TRIGGER Menu.FoodItem$InsertTrigger
ON Menu.FoodItem
AFTER INSERT
AS --Note, minimalist code for demo. Chapter 7 and Appendix B
   --have more details on complete trigger writing
BEGIN
                IF EXISTS (SELECT *
                                        FROM Inserted
                                        WHERE Description LIKE '%Yucky%')
        THROW 50000, 'No ''yucky'' food desired here',1;

END;
```

Now executing our batch with XACT_ABORT either ON or OFF returns

```
XACT_STATE  Description
----------- ------------------
-1          Doomed transaction
```

Wow, so many different outcomes! When building error handlers for your code, you need to either make sure that every possible outcome is handled or do a very good job of standardizing code in triggers, constraints, etc. to work only in one desired manner.

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, CREATE TABLE, ALTER INDEX, index reorganizations, etc.). 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; this is known as an *autocommit* transaction, as well as an *implicit* transaction. The SQL Server engine commits the transactions it starts for each statement-level transaction automatically, known as an autocommit.

WOW! eBook
www.wowebook.org

This is not the default behavior for some RDBMSs, so SQL Server gives us a setting to change this behavior: `SET IMPLICIT_TRANSACTIONS`. When this setting is turned on and the execution context isn't already within a transaction, `BEGIN TRANSACTION` is automatically (logically) executed when any of the following statements are executed: `INSERT`, `UPDATE`, `DELETE`, `SELECT` (when it touches a table), `TRUNCATE TABLE`, `DROP`, `ALTER TABLE`, `REVOKE`, `CREATE`, `GRANT`, `FETCH`, or `OPEN`. This means 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 and 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.

SQL Server Concurrency Methods

Until SQL Server 2005, there was only one concurrency method implemented in SQL Server. This was the pessimistic concurrency mechanism using locks. If a user had control of a row, a lock was placed, and all users who wanted incompatible use of the row (like two people can read a rows simultaneously, but only one could change the row, and readers have to wait, for example) were forced to wait. In 2005, a small taste of optimistic concurrency mechanisms was incorporated into the engine. Previous versions of data were written to `tempdb` to allow, when requested, a connection that would have been blocked from reading a row to see a version of the row that had been committed to be read. Writers could still block other users, and locks were still employed in all cases.

In SQL Server 2014, the new in-memory OLTP engine was implemented, which uses a complete optimistic concurrency mechanism, employing versions of rows rather than locks to manage concurrency. It is very different in many ways to what we have known of the engine for the past 20 years, which in most cases is a great thing. It is just different enough that one needs to understand what is going on first.

In this section I will introduce the settings that define how much one transaction can affect another, known as *isolation levels*, then I will explain and demonstrate how these work in pessimistic and optimistic concurrency enforcement.

What is interesting is that one transaction can mix different levels of concurrency implementation and locks. This will be particularly true when transactions enlist on-disk and in-memory OLTP tables, but can be true of any combination of table access, even within the same query by using isolation level hints.

Isolation Levels

In this section, we will identify, at an abstract level, how one transaction might affect another's view of the world. Of course, the safest method to provide consistency in operations would be to take exclusive access to the entire database, do your operations, and then release control. Then the next user would do the same thing. Although this was relatively common in early file-based systems, it isn't a reasonable alternative when you need to support 20,000 concurrent users.

WOW! eBook
www.wowebook.org

Isolation levels generally control how isolated one connection is from another, in terms of a few phenomenas that can occur. Take for instance, this query on a given connection:

```
BEGIN TRANSACTION;
UPDATE tableA
SET status = 'UPDATED'
WHERE tableAId = 'value';
```

On another connection, we have

```
BEGIN TRANSACTION;
INSERT tableA (tableAID, Status)
VALUES (100,'NEW');
```

Finally, we have

```
BEGIN TRANSACTION;
SELECT *
FROM   tableA;
```

Consider these are all executing simultaneously. They are all inside of transactions that say that they are not yet committed durably to the data structures. Will the `SELECT` statement see the new row? The changed row? The answer is "it depends." There are four specific phenomena that we need to deal with, and part of the answer depends on which query commits first, or rolls back.

The primary phenomena we are concerned with are

- *Dirty reads*: Seeing data that never may actually exist in a specific form. For example, when a second transaction selects a row that is in the process of being updated by a first transaction, the second transaction is reading data that has not been committed yet and may or may not be changed by the first transaction.
- *Phantom rows*: Seeing new rows in the results. Basically, when data is fetched once, the next time we fetch rows, any new rows are considered phantom rows.
- *Nonrepeatable reads*: Occur when data has been read in during a transaction, and when re-read the data has changed or has been deleted.
- *Expired rows*: Seeing data that no longer exists in the same form, but was consistent at a point in time.

While you can't specify at a given level "I will accept dirty reads, but not expired rows," there are five isolation levels that define an accepted set of phenomena for the different phenomena. The following list describes the isolation levels to adjust how one connection can affect another:

- `READ UNCOMMITTED`: Allows dirty reads, phantom rows, and nonrepeatable reads. Shows you data as it is, even if not fully committed, and as such it could change.
- `READ COMMITTED`: Allows phantom rows and nonrepeatable reads. So the same query could return different results in the same transaction, but you will not see uncommitted data.
- `REPEATABLE READ`: Allows phantom rows. New data is allowed to be added to the view of the query, but no data changes.

WOW! eBook
www.wowebook.org

- SERIALIZABLE: Like REPEATABLE READ, but no new rows can come into view of the query either. Like the name suggests, you are serializing access through the use of this data for the purpose of changing the data.
- SNAPSHOT: Lets you see expired rows when changes have been made during your transaction. Your view of the database will be constant, but things may have changed before you complete your transaction.

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 for a typical SQL Server query 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 due to giving you a consistent view of data. For example, say you are using the READ COMMITTED isolation level you read in all of the invoices, and before you have finished, invoices are deleted. You are reading in the invoice line items, and things don't quite match up. This scenario happens more often in things like ETL, but anything that easily happens in a large scale may still happen in a micro scale. If using the SNAPSHOT isolation level, you will see the data in a consistent state, even if it changes while you are reading it in.

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 is generally 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.

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 'Something is afoot'
        END
FROM    sys.dm_exec_sessions
WHERE  session_id = @@spid;
```

Unless you have already changed it, the default isolation level (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.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```


WOW! eBook
www.wowebook.org

■ **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.

The most common illustration of why transaction isolation is needed is called the *lost update*, as illustrated in Figure 11-1.

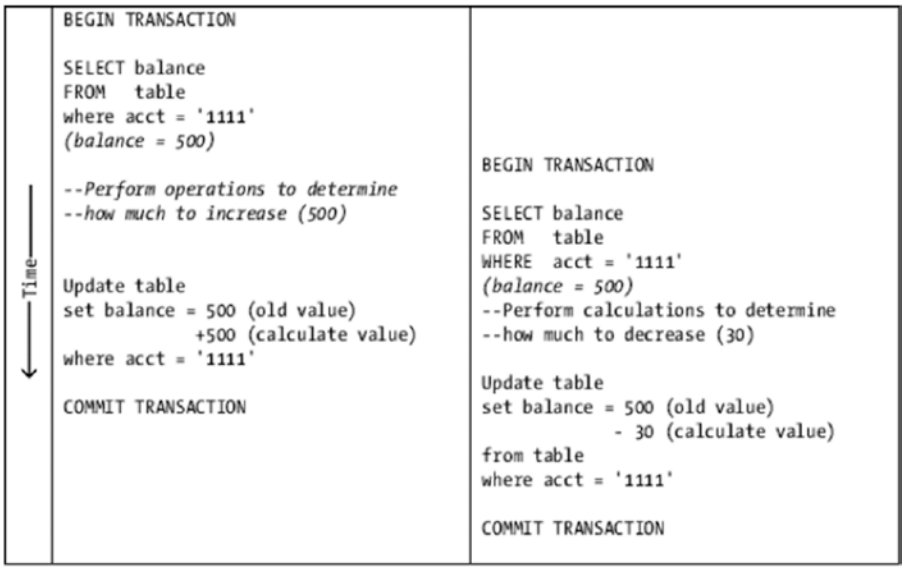


Figure 11-1. *A lost update illustration (probably one of the major inspirations for the other definition of multitasking: "screwing up everything simultaneously")*

The scenario in Figure 11-1 has 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 that is not going to make anyone happy. 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. In the following two sections, we will look at the two major paradigms for isolation connections.

Pessimistic Concurrency Enforcement

Pessimistic concurrency enforcement is done by taking control of a resource when using it. Sometimes the control is exclusive, like when writing data, or shared, when simply viewing data. This is achieved by using markers known as 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. How long the locks are held will be controlled by what the connection is doing as well as the isolation level.

WOW! eBook
www.wowebook.org

Every SQL Server process using an on-disk table applies locks to anything it is doing 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 its way. Minimally, a lock is always placed just to make sure that the database that is in use cannot be dropped.

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, your next trip to Disney World may be exclusively to the Haunted Mansion. (Ignoring locks isn't usually as messy as ignoring a train-crossing signal, but you could be creating the system that controls that warning signal.)

In this section, we 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

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 when a process is reading some data. 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 several 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 SQL Server uses each of them quite often. The list of locks types in Table 11-1 is not complete, but is a sample of the common types of locks we will encounter in modifying data in tables.

Table 11-1. *Lock Types*

Type of Lock	Granularity
Row or row identifier (RID)	A single row in a heap table.
Key range	A single value in an index. (Note that a clustered table is represented as an index in all physical structures.)
Key range	A range of key values (for example, to lock rows with values from A–M, even if no rows currently exist). Used for `SERIALIZABLE` isolation level.
Page	An 8KB index or data page.
HoBT	An entire heap or B-tree structure.
Table	An entire table, including all rows and indexes.
Application	A special type of lock that is user defined (will be covered in more detail later in this "Pessimistic Concurrency Enforcement" section)
Metadata	Metadata about the schema, such as catalog objects.
Database	The entire database.

WOW! eBook
www.wowebook.org

■ **Tip** Table 11-1 lists the lock types you will generally come in contact with 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.

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 based on the isolation level. 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 much to lock the resource. For example, consider a construction site. Workers are generally allowed onto the site, but civilians who are not part of the process not permitted. 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	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	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	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	Communicates to other processes that taking one of the previously listed modes might be necessary. It establishes a lock hierarchy with taken locks, allowing processes that are trying to take a lock on a resource (like a table), that there are other connections with locks at a lower level such as a page. You might see this mode as intent shared, intent exclusive, or shared with intent exclusive.
Schema	Used to lock the structure of an object when it's in use, so you cannot alter a structure (like a table) when a user is reading data from it.

WOW! eBook
www.wowebook.org

Each of these modes, coupled with the type/granularity, describes a locking situation. For example, an exclusive table lock would generally 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 that any additional users simply wait until all of the incompatible locks have been released.

Table 11-3 shows which types are compatible with which others.

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's pessimistic concurrency 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 sections, I'll demonstrate the locking in each of the isolation levels, which in pessimistic schemes determines how long locks are held, depending on the protection that the isolation level affords. 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.

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>)
```


WOW! eBook
www.wowebook.org

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:

- `PagLock`: Force 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 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).

Before moving to the examples, an important term that 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:

```
Msg 1205, Level 13, State 51, Line 242
Transaction (Process ID ??) 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, and if done correctly, will help you with dealing with optimistic concurrency issues).

WOW! eBook
www.wowebook.org

■ **Tip** Proper deadlock handling requires that you build your applications in such a way that you can easily tell the entire stack of queries sent. This is done by proper use of transactions and batches. 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 reality 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 with concurrent connections, 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 sounds simply horrible to users: "chosen as deadlock victim"!).

■ **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.

The `SNAPSHOT` isolation level does not pertain to pessimistic concurrency controls, as it lets you see previous versions of data with the expectation that nothing will change. Readers don't block readers, so the examples will generally be along the lines of "make a change to the state of the database in an open transaction, then see what another connection can see when executing a statement of some sort in another connection."

■ **Note** There's a file in the downloads that is a lock viewer that uses `sys.dm_tran_locks` to show you the locks held by a connection. If you are working through the examples, use this to see the locks held in given situations.

Isolation Levels and Locking

In the following sections, I will demonstrate how locks are taken and honored in the following isolation levels:

- `READ UNCOMMITTED`
- `READ COMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`

WOW! eBook
www.wowebook.org

In the next sections on pessimistic and optimistic concurrency enforcement, I'll briefly discuss the different isolation levels and demonstrate how they work using the following tables (and alternate versions using in-memory OLTP):

```
CREATE SCHEMA Art;
GO
CREATE TABLE Art.Artist
(
    ArtistId int CONSTRAINT PKArtist PRIMARY KEY
    ,Name varchar(30) --no key on value for demo purposes
    ,Padding char(4000) default (replicate('a',4000)) --so all rows not on single page
);
INSERT INTO Art.Artist(ArtistId, Name)
VALUES (1,'da Vinci'),(2,'Micheangelo'), (3,'Donatello'),
       (4,'Picasso'),(5,'Dali'), (6,'Jones');
GO
CREATE TABLE Art.ArtWork
(
    ArtWorkId int CONSTRAINT PKArtWork PRIMARY KEY
    ,ArtistId int NOT NULL
           CONSTRAINT FKArtwork$wasDoneBy$Art_Artist REFERENCES Art.Artist (ArtistId)
    ,Name varchar(30)
    ,Padding char(4000) DEFAULT (REPLICATE('a',4000)) --so all rows not on single page
    ,CONSTRAINT AKArtwork UNIQUE (ArtistId, Name)
);
INSERT Art.Artwork (ArtworkId, ArtistId, Name)
VALUES (1,1,'Last Supper'),(2,1,'Mona Lisa'),(3,6,'Rabbit Fire');
```

READ UNCOMMITTED

A connection in READ UNCOMMITTED does not take locks on resources used (other than the database it is in), and does not honor locks by other connections other than metadata locks, such as a table that is in the process of being created. As mentioned, you still get locks on any data you modify, which is true of all isolation levels.

READ UNCOMMITTED is a great tool when you need to see the state of a data operation in-flight. Like if a transaction is loading a million rows into a table, instead of waiting for the transaction to complete, you can see the changes to the table. A second use of READ UNCOMMITTED is when you are the admin user and need to look at a production resource without blocking others. For example, say we add the following row to the Art.Artist table, but we do not commit the transaction:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; --this is the default, just
                                                --setting for emphasis
BEGIN TRANSACTION;
INSERT INTO Art.Artist(ArtistId, Name)
VALUES (7, 'McCartney');
```


WOW! eBook
www.wowebook.org

Then on a different connection we execute

```
--CONNECTION B
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT ArtistId, Name
FROM Art.Artist
WHERE Name = 'McCartney';
```

This will sit and not complete, because the row has not been committed. However, if we change this to READ UNCOMMITTED:

```
--CONNECTION B
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT ArtistId, Name
FROM Art.Artist
WHERE Name = 'McCartney';
```

we will immediately get back:

```
ArtistId    Name
----------- ------------------------------
7           McCartney
```

The problem is, when the process on Connection A decides to rollback their connection:

```
--CONNECTION A
ROLLBACK TRANSACTION;
```

Go back and execute that statement again and it is gone. Whether this is a problem or not is a matter for what the first half of the book was about: requirements. Too often though, if some process were to uppercase all of the names in this table, and another user saw that and called the service desk, by the time they checked, all would back to normal. While this might be a fun way to gaslight your service desk, not good practice.

Being able to see locked data is quite valuable, especially when you're in the middle of a long-running process, 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.

■ **Caution** Ignoring locks using `READ UNCOMMITTED` is almost never the right 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 the `SNAPSHOT` isolation level or the `READ COMMITTED SNAPSHOT` database setting, which I will cover in the "Optimistic Concurrency Enforcement" section, can give you sort of the same performance without reading dirty data.

WOW! eBook
www.wowebook.org

READ COMMITTED

READ COMMITTED is the default isolation level as far as SQL Server is concerned, and as the name states, it only 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 accessing the resource. (Hence, if you are using 1,000 resources, it may take one lock, use a resource, release the lock, and access the next resource.) 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. As the data architect, you need to answer the questions "What if data changes?" and see if this matters.

As an example, we will start a transaction, and then view data in the Art.Artist table:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;
SELECT ArtistId, Name FROM Art.Artist WHERE ArtistId = 7;
```

No rows are returned. Locks are only taken on resources as they are actually being used, so if you could run the lock viewer fast enough, you would just see one or so locks at a time. Next, on a separate connection, we will add a new row to the table, and commit it (I demonstrated in the previous section that we would be blocked from viewing the row in READ COMMITTED, since the row was in a dirty state):

```
--CONNECTION B
INSERT INTO Art.Artist(ArtistId, Name)
VALUES (7, 'McCartney');
```

Going back to Connection A, we reselect the data, still in the same transaction:

```
--CONNECTION A
SELECT ArtistId, Name FROM Art.Artist WHERE ArtistId = 7;
```

It now returns:

```
ArtistId    Name
----------- ------------------------------
7           McCartney
```

Update the row on Connection B:

```
--CONNECTION B
UPDATE Art.Artist SET Name = 'Starr' WHERE ArtistId = 7;
```

Then view and commit the transaction on the row again:

```
--CONNECTION A
SELECT ArtistId, Name FROM Art.Artist WHERE ArtistId = 7;
COMMIT TRANSACTION;
```


WOW! eBook
www.wowebook.org

```
ArtistId    Name
----------- ------------------------------
7           Starr
```

The bottom line is that this isolation level is great for most cases, particularly when you are doing a single query. However, as the number of statements increases, it becomes less desirable. For example, if part of your process had been to be to look to make sure that an artist named "McCartney" existed in the table. A transaction was started, to contain the operation, and the row was seen to exist. But after that the row changed to "Starr," so your query to check for that row has been invalidated.

The other process may have done its own checks to make sure it could update the rows, which may have been prevented by some step in your process that has not yet completed. Which is to say this: concurrency can be complex, and it is very hard to test. The timing of the aforementioned scenario might be just microseconds, and it could never happen. It certainly is unlikely to occur on the developer's workstation, and unless you do massive amounts of testing, it may not occur in your test system either. But like Murphy's Law states: "Anything that can go wrong, will go wrong," so you need to program defensively and consider "What if this happens?" testing using extra `BEGIN TRANSACTION` and often `WAITFOR` statements can be very helpful in slowing down time enough to cause collisions that would rarely if ever occur.

The next two isolation levels will be particularly useful in dealing with data integrity checks, as we can hold locks on the rows we have accessed longer than just to make sure they are not changed as we return them.

REPEATABLE READ

The `REPEATABLE READ` isolation level includes protection from data being changed or 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 are 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 ArtistId, Name FROM Art.Artist WHERE ArtistId >= 6;
```

This returns

```
ArtistId    Name
----------- ------------------------------
6           Jones
7           Starr
```

Then on another connection, execute the following:

```
--CONNECTION B
INSERT INTO Art.Artist(ArtistId, Name)
VALUES (8, 'McCartney');
```


WOW! eBook
www.wowebook.org

If you go back to CONNECTION A and execute the SELECT again, you will see three rows. But if you try to delete the row with ArtistId = 6:

```
--CONNECTION B
DELETE Art.Artist
WHERE  ArtistId = 6;
```

you will find that you are blocked. Connection A holds a shared key lock on the row number 6 (and now 7 and 8 also since you have seen them), so that row cannot be deleted or updated, which would require an incompatible exclusive lock. Cancel the query on Connection B, and then COMMIT the transaction on Connection A to continue.

REPEATABLE READ isolation level is perfect for the cases where you want to make sure that some data still exists when your process has completed (like if you were implementing a foreign key–type check in code).

It is not ideal for the situation where you want to maintain a balance (what if a negative value is inserted?) or where you want to limit the cardinality of a relationship (new rows are fine). We will explore how to deal with that situation in the next section.

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 often 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 Art.Artist
```

For the duration of the execution of this query (and longer if in an explicit transaction), due to a range lock being taken on all keys in the table, no other user would be able to modify or add rows to the table until all rows have been returned and the transaction it was executing within has completed.

■ **Note** Be careful. I said, "No other user would be able to modify or add rows to 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, and then write them later. We will look at techniques to avoid this issue later in this chapter as we discuss techniques to write our code to deal with concurrency in mind.

If lots of users are viewing data in the table in the SERIALIZABLE isolation level, 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. Execute this statement on a connection to simulate a user with a table locked:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;
SELECT ArtistId, Name FROM Art.Artist;
```


WOW! eBook
www.wowebook.org

Then, try to add a new row to the table:

```
--CONNECTION B
INSERT INTO Art.Artist(ArtistId, Name)
VALUES (9, 'Vuurmann'); --Misspelled on purpose. Used in later example
```

Your insert is blocked. Commit the transaction on `CONNECTION A`:

```
--CONNECTION A
COMMIT TRANSACTION;
SELECT ArtistId, Name FROM Art.Artist;
```

This unblocks `CONNECTION B`, and you will see that the contents of the table are now the following:

```
ArtistId    Name
----------- ------------------------------
1           da Vinci
2           Micheangelo
3           Donatello
4           Picasso
5           Dali
6           Jones
7           Starr
8           McCartney
9           Vuurmann
```

It is important to be careful with the `SERIALIZABLE` isolation level. The name tells you that you will have single-threaded access to resources in this table, particularly when writing data to the table. I can't stress enough that while multiple readers can read the same data, no one can change 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.

Interesting Cases

In this section I will present a few interesting cases that are more than simple isolation cases, including foreign keys and application locks that can help us implement a critical section of code.

Locking and Foreign Keys

One of the interesting cases when working with locking involves foreign keys. Consider our tables: `Art.Artist` and the child table, `Art.Artwork`. When you insert a new row into the `Art.Artwork` table, it must check the `Art.Artist` table to see if the key exists:

```
--CONNECTION A
BEGIN TRANSACTION;
INSERT INTO Art.ArtWork(ArtWorkId, ArtistId, Name)
VALUES (4,9,'Revolver Album Cover');
```


WOW! eBook
www.wowebook.org

Looking at the locks held, you will only see locks on the `ArtWork` table. So it looks like there is no guarantee of the row for `ArtistId = 9` being deleted. This is true in terms of locked resources, but foreign keys protect both ways. So if on a different connection the following code is executed while this transaction is still open:

```
--CONNECTION B
DELETE FROM Art.Artist WHERE ArtistId = 9;
```

it will be blocked from executing because while there are no rows currently it needs to check is locked. So the new row will determine whether or not the artist can be deleted. On Connection A, execute

```
-- CONNECTION A
COMMIT TRANSACTION;
```

Then, Connection B will complete with a delete reference error, since it has child rows existing. However, to show that the `Art.Artist` row is not locked, you can update it:

```
--CONNECTION A
BEGIN TRANSACTION;
INSERT INTO Art.ArtWork(ArtWorkId, ArtistId, Name)
VALUES (5,9,'Liverpool Rascals');
```

Now, on connection B execute

```
--CONNECTION B
UPDATE Art.Artist
SET  Name = 'Voorman'
WHERE artistId = 9;
```

It will succeed. Then back on Connection A, roll back the transaction, so that the new row is not added:

```
--CONNECTION A
ROLLBACK TRANSACTION;
SELECT * FROM Art.Artwork WHERE ArtistId = 9;
```

You will see just the one row returned. So what about cascading foreign keys? There are two different cases. If no child row exists, instead of taking no locks on the child table, an intent exclusive lock is placed on the table. This would keep anyone else from taking an exclusive lock on the table, but would otherwise allow usage. If a user tries to insert or update a row that uses the deleted key, the foreign key's checking would bump up against the locked row just as before.

If rows do exist in the child to be deleted, they will be exclusively locked just as if you did a delete, and a range exclusive lock is set to prevent anyone from inserting a row that might match the parent. In the downloads, I alter the foreign key constraint and include a few deletes for seeing this occur.

Foreign keys do slow down some operations in the execution of database code, but fortunately, when using a locking scheme, that effect is not necessarily long felt, even when a row that references another table

WOW! eBook
www.wowebook.org

Application Locks

SQL Server does have a built-in method you can use to implement a form of pessimistic locking: SQL Server *application locks*. 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. However, for the cases where you need to single thread code for some reason, applocks are a great resource.

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 this to check the mode of the application lock.
- `APPLOCK_TEST`: Use this 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 a logical invoice object. We'll set it as an exclusive lock so no other user can touch it. In one connection, we run the following code (note, applocks are database specific, so you will need to be in the same database on Connections A and B). By default, applocks are associated with a transaction, so you will need to be in an explicit transaction. You can specify a parameter of `@LockOwner` of `Session` and you will not need a transaction, but it will be around until you use `sp_releaseAppLock` or end the connection, which can be more difficult to manage.

```
--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. Other possible output is shown here:

```
Value       Result
----------- ------------------------------
          1 Lock granted after waiting for other locks
         -1 Timed out
         -2 Lock request cancelled
         -3 Deadlock victim
        999 Other error
```

You can see the type of applock taken by using `APPLOCK_MODE()`:

```
SELECT APPLOCK_MODE('public','invoiceId=1');
```


WOW! eBook
www.wowebook.org

This returns the mode the lock was taken in, in our case `Exclusive`. 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. The other output is `1`, indicating the lock could be granted. `APPLOCK`s can be a great resource for building locks that are needed to implement locks that are "larger" than just SQL Server objects. 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.

■ **Tip** You can use application locks to implement more than just exclusive locks using different lock modes, 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.

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.

WOW! eBook
www.wowebook.org

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 SCHEMA Demo;
GO
CREATE TABLE Demo.Applock
(
    ApplockId int CONSTRAINT PKApplock PRIMARY KEY,
                              --the value that we will be generating
                              --with the procedure
    ConnectionId int,         --holds the spid of the connection so you can
                              --who creates the row
    InsertTime datetime2(3) DEFAULT (SYSDATETIME()) --the time the row was created, so
                                                    --you can see the progression
);
```

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 Demo.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;
```


WOW! eBook
www.wowebook.org

```
    --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 Demo.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 Demo.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.
    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 XACT_STATE() <> 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 Demo.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 Surface Pro 4, i7, in an 8GB VM 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 slightly more time than the original time, without any clashes whatsoever.

WOW! eBook
www.wowebook.org

To solidify the point that every connection has to follow the rules, turn off application locks on only 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 any delay.

This is not the only method of implementing the solution to the incrementing values problem. Another 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.

Optimistic Concurrency Enforcement

Whereas pessimistic concurrency is based on holding locks on resources, expecting multiple users utilizing resources for noncompatible reasons, optimistic concurrency is just the opposite. In optimistic concurrency control, we expect no overlap in use and optimize accordingly, giving users access to previous versions of resources that have been or are being modified. The outcome is a tremendous reduction in most waiting times, though it can be costlier when concurrency collisions do occur.

Starting with SQL Server 2005, there was one model of optimistic concurrency enforcement in use that was comingled with the locking we saw in the pessimistic locking scheme. This partial implementation did amazing things for performance, but still makes use of locks to deal with write contention. As such, it was not really used tremendously in total, but there was one feature that can do wonders to your lock contention issues.

In SQL Server 2014, Microsoft added the in-memory OLTP engine, which has a completely different concurrency model called Multi-Value Concurrency Control (MVCC), which is based completely on versions with no locks involved, other than locks purely for schema stability purposes. In Chapter 10, the structural differences were enumerated, but in this chapter we will see them in use.

In the following sections, I will cover the on-disk implementation of optimistic concurrency, followed by coverage of the in-memory OLTP engine considerations for concurrency.

Optimistic Concurrency Enforcement in On-Disk Tables

There are two main topics to look at when it comes to on-disk optimistic concurrency enforcement. The first is `SNAPSHOT` isolation level, which affects your connection at a transaction level, and the database setting `READ COMMITTED SNAPSHOT`, which affects your queries at a statement level (much like `READ COMMITTED` does now, you can get different results on different executions in the same transaction). For `SNAPSHOT` isolation level, two executions of the same query will return the same results, unless you change the table in your connection (which can lead to issues, as we will see).

WOW! eBook
www.wowebook.org

SNAPSHOT Isolation Level

SNAPSHOT isolation lets you read the data as it was when the transaction started, regardless of any changes. No matter how much the data changes in the "real" world, your view stays the same. This makes it impossible to do things like check to see if a row exists in another table, since you will not be able to see it in your transaction.

The largest downside is the effect SNAPSHOT isolation can have on performance if you are not prepared for it. This history data for on-disk tables 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 a 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 to view corporate performance in the middle of the busiest part of the day, that query will never be blocked, and no other users will get stuck behind it either. The better news is that there will be no cases where a mistaken $10 million entry that one of the data-entry clerks added to the data that the check constraint/trigger 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 Chapter11
     SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Now, the SNAPSHOT isolation level is available for queries.

■ **Reminder** The SNAPSHOT isolation level uses copies of changed data placed into tempdb. Because of this, you should make sure that your tempdb is set up optimally. For more on setting up hardware, consider checking out the web site www.sql-server-performance.com.

Let's look at an example. On the first connection, start a transaction and select from the Art.Artist table we used back in the "Pessimistic Concurrency Enforcement" section:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
SELECT ArtistId, Name FROM Art.Artist;
```


WOW! eBook
www.wowebook.org

This returns the following results:

```
ArtistId    Name
----------- ------------------------------
1           da Vinci
2           Micheangelo
3           Donatello
4           Picasso
5           Dali
6           Jones
7           Starr
8           McCartney
9           Vuurmann
```

On a second connection, run the following:

```
--CONNECTION B
INSERT INTO Art.Artist(ArtistId, Name)
VALUES (10, 'Disney');
```

This executes with no waiting. Going back to Connection A, without ending the transaction and 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 Art.Artist
WHERE  ArtistId = 3;
```

This doesn't have to wait either. Going back to the other connection again, nothing has changed:

```
--CONNECTION A
SELECT ArtistId, Name FROM Art.Artist;
```

This still returns the same nine rows. Commit the transaction, and check the differences.

```
ArtistId    Name
----------- ------------------------------
1           da Vinci
2           Micheangelo
4           Picasso
5           Dali
6           Jones
7           Starr
8           McCartney
9           Voormann
10          Disney
```


WOW! eBook
www.wowebook.org

So what about modifying data in SNAPSHOT isolation level? If no one else has modified the row, you can make any change:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;

UPDATE Art.Artist
SET    Name = 'Duh Vinci'
WHERE  ArtistId = 1;

ROLLBACK;
```

But, if you have two connections competing and fetching/modifying the same row, you may get blocking. For example, say connection B has a lock from a transaction in a non-SNAPSHOT isolation level:

```
--CONNECTION B
BEGIN TRANSACTION

UPDATE Art.Artist
SET    Name = 'Dah Vinci'
WHERE  ArtistId = 1;
```

Then connection A updates it:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

UPDATE Art.Artist
SET    Name = 'Duh Vinci'
WHERE  ArtistId = 1;
```

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 the SNAPSHOT isolation level. When making changes to data in the SNAPSHOT isolation level, the changes still take locks, which will block any non-SNAPSHOT isolation transaction connection just like before. However, what is very different is how it deals with two SNAPSHOT isolation level connections modifying the same resource. There are two cases. First, if the row has not been yet cached, you can get blocked. For example, (after rolling back the data changes on both connections), start a new transaction, and update `ArtistId 1` to `Duh Vinci`:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;

UPDATE Art.Artist
SET    Name = 'Duh Vinci'
WHERE  ArtistId = 1;
```


WOW! eBook
www.wowebook.org

Then, a second user changes the value, to `Dah Vinci`:

```
--CONNECTION B
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRANSACTION;

UPDATE Art.Artist
SET    Name = 'Dah Vinci'
WHERE  ArtistId = 1;
```

You will get blocked, something that is not common to optimistic locking schemes. In this next (quite typical) case (again, after rolling back the transactions from the previous example), connection A caches the row, connection B updates, and connection A tries to update it also:

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
SELECT *
FROM   Art.Artist;
```

Then execute the following on Connection B, to change the data. It is not in an explicit transaction.

```
--CONNECTION B
UPDATE Art.Artist
SET    Name = 'Dah Vinci'
WHERE  ArtistId = 1;
```

Now when you try to update the row that you think exists:

```
--CONNECTION A
UPDATE Art.Artist
SET    Name = 'Duh Vinci'
WHERE  ArtistId = 1;
```

the following error message rears its ugly head because this row has been deleted by a different connection:

```
Msg 3960, Level 16, State 3, Line 586
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot
isolation to access table 'Art.Artist' directly or indirectly in database 'Chapter11'
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.
```

For the most part, due to the way things work with SNAPSHOT intermingled with the pessimistic concurrency engine, it can be complex (or perhaps annoying is a better term) to work with. There is a key phrase in the error message: "retry the transaction." This was a theme from deadlocks, and will be a major theme in the in-memory implementation of MVCC.

WOW! eBook
www.wowebook.org

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. It is a very powerful tool for performance tuning, but it should be noted that it changes the way the entire database processes READ COMMITTED transactions, and because data may be changing, it will behoove you to think deeply about how the code you write to check data integrity may be affected by changes that are in flight when you do your checks.

The important part to understand is that this works on a "statement" level and not a "transaction" level. 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
ALTER DATABASE Chapter11
    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 wait for any in-process 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).

For places where you might need more safety, use 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.

■ **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.

WOW! eBook
www.wowebook.org

As an example, after setting the database setting, on Connection B we will add a new row in a transaction, but not commit it. On Connection A, we will search for the rows of the table.

```
--CONNECTION A
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRANSACTION;
SELECT ArtistId, Name FROM Art.Artist;
```

This returns the table as we had in the last section:

```
ArtistId    Name
----------- ------------------------------
1           da Vinci
2           Micheangelo
4           Picasso
5           Dali
6           Jones
7           Starr
8           McCartney
9           Voormann
10          Disney
```

```
--CONNECTION B
BEGIN TRANSACTION;
INSERT INTO Art.Artist (ArtistId, Name)
VALUES  (11, 'Freling' )


--CONNECTION A
SELECT ArtistId, Name FROM Art.Artist;
```

Next, we update all of the rows, including the one we just added:

```
--CONNECTION B (still in a transaction)
UPDATE Art.Artist
SET  Name = UPPER(Name);

--CONNECTION A
SELECT ArtistId, Name FROM Art.Artist;
```

This still returns the same data. We could delete all of the rows, and nothing would change. Commit the transaction on B, then go back and look at the data on Connection A:

```
--CONNECTION B
COMMIT;

--CONNECTION A
SELECT ArtistId, Name FROM Art.Artist;
COMMIT;
```


WOW! eBook
www.wowebook.org

We see the changes in the data, then commit the transaction so we don't mess up the next demos.

```
ArtistId    Name
----------- ------------------------------
1           DAH VINCI
2           MICHEANGELO
4           PICASSO
5           DALI
6           JONES
7           STARR
8           MCCARTNEY
9           VOORMANN
10          DISNEY
11          FRELING
```

Optimistic Concurrency Enforcement in In-Memory OLTP Tables

In the previous chapter, I discussed the differences between the on-disk and in-memory physical structures. In this chapter, we will take this farther and look at how code will work. While the in-memory OLTP tables are generally meant to work from the same code as their on-disk cousins, the internals, especially as it deals with concurrency, is quite different.

For an example, consider a table that has an identity column and one that holds the country. When we first start the server, and the in-memory columns are loaded, we have the structures like the following at timestamp 0:

```
TableNameId Country              OtherColumns...
----------- -------------------- ---------------------------
1           USA                  Values
2           USA                  Values
3           Canada               Values
```

Next, `CONNECTION A` starts a transaction at timestamp 50. `CONNECTION B` starts a transaction and updates the row with `TableNameId = 2` to have the `Country = 'Canada'`. As we discussed in Chapter 10, there can only be the one "dirty" row, so no other user could try to update that same row while `CONNECTION B` is in the process. At timestamp 100, `CONNECTION B` commits their transaction, but `CONNECTION A`'s connection to the timestamp 50 version is still in use. So the live table looks like:

```
TableNameId Country              OtherColumns...
----------- -------------------- ---------------------------
1           USA                  Values
2           Canada               Values
3           Canada               Values
```


WOW! eBook
www.wowebook.org

Internally, there are four rows, as both views of the table that we have seen so far can been viewed by one or more users. So we have the basic structure shown in Figure 11-2.

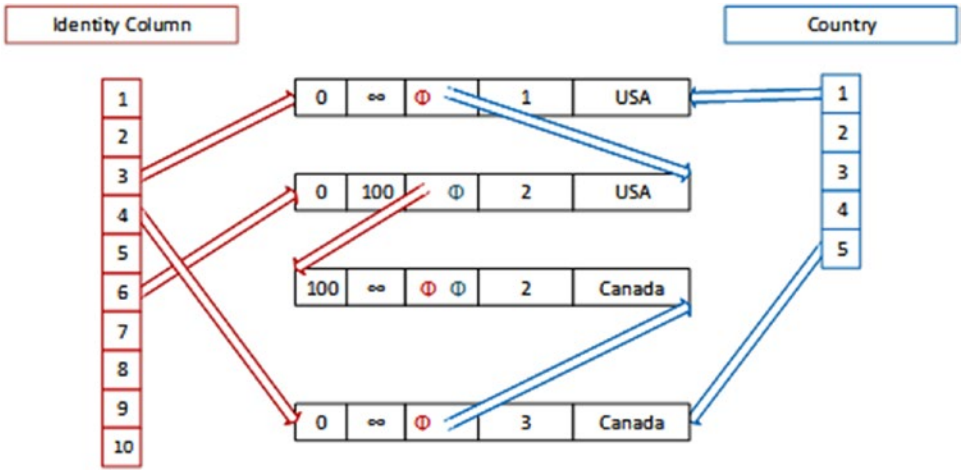


Figure 11-2. *Sample index structure at timestamp 100 and later*

Now if the user searches on `CONNECTION A`, still stuck in timestamp 50 time, searching for `Country = 'USA'` will return two rows, and after timestamp 100, it will return one row. In these structures, `CONNECTION A` will never be able to see what happens after timestamp 50, providing isolation that makes sure a user cannot change their view of the database. There are three isolation levels supported: `SNAPSHOT`, `REPEATABLE READ`, and `SERIALIZABLE`.

For reading data, this is done (for `REPEATABLE READ`, and `SERIALIZABLE` as `SNAPSHOT` allows any amount of change) at commit time to see if data has been modified or created that would change the view of the data in an incompatible way. So if a row was deleted, it would fail at `COMMIT` time with an error that says there has been an isolation level issue.

Not being able to see what has occurred on other connections means that you will likely need to use the isolation levels other than `SNAPSHOT` more frequently. For example, say you want to check that a row of a certain type exists, it is impossible to know if the row is deleted, or if one is inserted. However, unlike pessimistic locking scenarios, when employing optimistic versions, the impact on other connections is likely to be less, since you will not be preventing them from using their view of the data. It will make certain sorts of implementations very undesirable, however. For example, implementing a queue without locks could take lots of iterations in code, do all of the work, and then rolling back when you discover someone else has committed their changes.

For writing data, things get more interesting. As noted in Chapter 10, there is only one "dirty" version of a row. When two connections try to modify the same row, it is physical resource collision that gives an immediate error. If it is not a physical resource collision (for example, two connections insert duplicate values), the index pointers will not yet be set until `COMMIT`, when they will become a physical resource collision and you will get an error. In the following sections, I will demonstrate each of these cases to make it more clear.

A term that should be introduced here is *cross container*. The on-disk tables and in-memory table are in the same database, but in different containers. Once you access resources in one container or another it is part of a transaction. When you access resources in both containers (like accessing an in-memory table from interpreted T-SQL), it is considered cross container. So the transaction timestamp will start once you access an in-memory resource when running in interop mode (which all of the examples in this chapter will be).

WOW! eBook
www.wowebook.org

For the examples, I will be using the same basic tables as we had in the previous sections on pessimistic concurrency control, but the schema will indicate that it is an in-memory OLTP object (note that this example only works on SQL Server 2016, and not the 2014 version of in-memory OLTP due to multiple uniqueness constraints and foreign keys):

```
--The download will include the code to add an in-memory filegroup

CREATE SCHEMA Art_InMem;
GO
CREATE TABLE Art_InMem.Artist
(
    ArtistId int CONSTRAINT PKArtist PRIMARY KEY
                                        NONCLUSTERED HASH  WITH (BUCKET_COUNT=100)
    ,Name varchar(30) --no key on value for demo purposes, just like on-disk example
    ,Padding char(4000) --can't use REPLICATE in in-memory OLTP, so will use in INSERT

) WITH ( MEMORY_OPTIMIZED = ON );

INSERT INTO Art_InMem.Artist(ArtistId, Name,Padding)
VALUES (1,'da Vinci',REPLICATE('a',4000)),(2,'Micheangelo',REPLICATE('a',4000)),
       (3,'Donatello',REPLICATE('a',4000)),(4,'Picasso',REPLICATE('a',4000)),
           (5,'Dali',REPLICATE('a',4000)), (6,'Jones',REPLICATE('a',4000));
GO

CREATE TABLE Art_InMem.ArtWork
(
    ArtWorkId int CONSTRAINT PKArtWork PRIMARY KEY
                                          NONCLUSTERED HASH  WITH (BUCKET_COUNT=100)
    ,ArtistId int NOT NULL
        CONSTRAINT FKArtwork$wasDoneBy$Art_Artist REFERENCES Art_InMem.Artist (ArtistId)
    ,Name varchar(30)
    ,Padding char(4000) --can't use REPLICATE in in-memory OLTP, so will use in INSERT
    ,CONSTRAINT AKArtwork UNIQUE NONCLUSTERED (ArtistId, Name)
) WITH ( MEMORY_OPTIMIZED = ON );
INSERT Art_InMem.Artwork (ArtworkId, ArtistId, Name,Padding)
VALUES (1,1,'Last Supper',REPLICATE('a',4000)),(2,1,'Mona Lisa',REPLICATE('a',4000)),
       (3,6,'Rabbit Fire',REPLICATE('a',4000));
```

We need to cover a few things before moving on. First, in-memory OLTP objects cannot be involved in a transaction that crosses database boundaries unless that database is `tempdb` or `master` (`master` in a read-only manner). So the following is prohibited:

```
INSERT INTO InMemTable
SELECT *
FROM DBOtherThanTempdb.SchemaName.Tablename;
```

This can make things interesting in some situations, particularly when building a demo database. I usually have a copy of the data held in a separate database so I can reload. But to move from a different database in Transact-SQL, you need to move the data to `tempdb` first.

WOW! eBook
www.wowebook.org

Second, when accessing memory-optimized tables in an explicit transaction, you generally need to specify the isolation level. For example, this works:

```
SELECT ArtistId, Name
FROM   Art_Inmem.Artist;
```

But the following:

```
BEGIN TRANSACTION;
SELECT ArtistId, Name
FROM   Art_Inmem.Artist;
COMMIT TRANSACTION;
```

will return the following error:

```
Msg 41368, Level 16, State 0, Line 667
Accessing memory optimized tables using the READ COMMITTED isolation level is supported
only for autocommit transactions. It is not supported for explicit or implicit
transactions. Provide a supported isolation level for the memory optimized table using a
table hint, such as WITH (SNAPSHOT).
```

There are still some limitations on what can be mixed also. For example, if you try to access the table when the connection isolation level is `REPEATABLE READ`:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION;
SELECT ArtistId, Name
FROM   Art_Inmem.Artist WITH (REPEATABLEREAD);
COMMIT TRANSACTION;
```

you get the following:

```
Msg 41333, Level 16, State 1, Line 684
The following transactions must access memory optimized tables and natively compiled
modules under snapshot isolation: RepeatableRead transactions, Serializable transactions,
and transactions that access tables that are not memory optimized in RepeatableRead or
Serializable isolation.
```

You need to specify

```
BEGIN TRANSACTION;
SELECT ArtistId, Name
FROM   Art_Inmem.Artist WITH (SNAPSHOT);
COMMIT TRANSACTION;
```


WOW! eBook
www.wowebook.org

Or for the simple case where you want it to be SNAPSHOT isolation level unless specified, you can use

```
ALTER DATABASE LetMeFinish
  SET MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT ON;
```

I will not do this in my sample code, simply to make it clearer what isolation level I am using.

The following sections will first show how readers of data deal with chances to data by a different connection in SNAPSHOT, REPEATABLE READ, and SERIALIZABLE isolation levels, and then show the different ways that one writer is isolated from another writer. Unlike the pessimistic strategy used for on-disk tables that almost always results in a blocking situation, there are several interesting differences in the way one writer may affect the action of another.

Note that there are a few differences to how isolation is done using native code. I will cover this in Chapter 13 when I cover code design practices.

SNAPSHOT Isolation Level

When you read tables in SNAPSHOT isolation level, just like in the on-disk version, no matter what happens in another connection, things just look to you like nothing ever changes.

To demonstrate, on CONNECTION A, we will start a transaction, and then on CONNECTION B, we will create a new row:

```
--CONNECTION A
BEGIN TRANSACTION;

--CONNECTION B
INSERT INTO Art_InMem.Artist(ArtistId, Name)
VALUES (7, 'McCartney');
```

Now on CONNECTION A, we look at the rows where ArtistId > 5:

```
--CONNECTION A
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId >= 5;
```

This returns the following, which matches the current state in the database:

```
ArtistId    Name
----------- ------------------------------
7           McCartney
5           Dali
6           Jones
```


WOW! eBook
www.wowebook.org

While the overall transaction container started with the BEGIN TRANSACTION started earlier, the in-memory transaction container starts when you cross the container barrier and then enlists in-memory resources. We started with three rows in Artist, so let's add a row to both tables on CONNECTION B, and delete ArtistId = 6 and see what we can see:

```
--CONNECTION B
INSERT INTO Art_InMem.Artist(ArtistId, Name)
VALUES (8, 'Starr');

INSERT INTO Art_InMem.Artwork(ArtworkId, ArtistId, Name)
VALUES (4,7,'The Kiss');

DELETE FROM Art_InMem.Artist WHERE ArtistId = 5;
```

read all of the rows from Art_InMem.Artist where the ArtistId is 5 or greater. Back on CONNECTION A, if we read the data in the same transaction:

```
--CONNECTION A
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId >= 5;

SELECT COUNT(*)
FROM  Art_InMem.Artwork WITH (SNAPSHOT);
```

our original view still persists:

```
ArtistId    Name
----------- ------------------------------
7           McCartney
5           Dali
6           Jones
-----------
3
```

If we COMMIT or ROLLBACK (we haven't made any changes, so the net effect is the same) the transaction on CONNECTION A, our view of the data changes to what is current:

```
--CONNECTION A
COMMIT;

SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId >= 5;

SELECT COUNT(*)
FROM  Art_InMem.Artwork WITH (SNAPSHOT);
```


WOW! eBook
www.wowebook.org

Our view is consistent with the current reality:

```
ArtistId    Name
----------- ------------------------------
7           McCartney
8           Starr
6           Jones
-----------
4
```

REPEATABLE READ Isolation Level

`REPEATABLE READ` isolation level provides the same protection using optimistic versions as it does with the pessimistic locking scheme. The difference is how you notice the change. Just like in the previous example, we will start with `CONNECTION A` reading in a set of rows, then `CONNECTION B` making a change. We will start with `CONNECTION B` adding a row, which will cause a phantom row for `CONNECTION B`, which is acceptable in `REPEATABLE READ`.

```
--CONNECTION A
BEGIN TRANSACTION;
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (REPEATABLEREAD)
WHERE ArtistId >= 8;
```

This returns the one row we knew existed:

```
ArtistId    Name
----------- ------------------------------
8           Starr
```

Now, on `CONNECTION B`, we insert a new row, not in an explicit transaction.

```
--CONNECTION B
INSERT INTO Art_InMem.Artist(ArtistId, Name)
VALUES (9,'Groening');
```

Then we go look at the row in the transaction context, and commit the transaction and look again:

```
--CONNECTION A
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId >= 8;
COMMIT;
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId >= 8;
```


WOW! eBook
www.wowebook.org

The first result set will return the same as before, but the second will have two rows:

```
ArtistId    Name
----------- ------------------------------
8           Starr
9           Groening
```

This time we will delete one of the rows after we fetch it on CONNECTION A:

```
--CONNECTION A
BEGIN TRANSACTION;
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (REPEATABLEREAD)
WHERE ArtistId >= 8;

--CONNECTION B
DELETE FROM Art_InMem.Artist WHERE ArtistId = 9; --Not because I don't love Matt!
```

Then access the rows before and after committing the transaction:

```
--CONNECTION A
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId >= 8;
COMMIT;
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId >= 8;
```

We get two rows back, and then we get an error message:

```
Msg 41305, Level 16, State 0, Line 775
The current transaction failed to commit due to a repeatable read validation failure.
```

So if our transaction had been open for 10 minutes, or 10 hours, the work that had been done is gone, and the entire transaction rolled back. Keeping transactions short can mitigate the cost of this, but it is certainly something to understand.

SERIALIZABLE Isolation Level

SERIALIZABLE isolation level behaves just like REPEATABLE READ, except that you get an error whether you are inserting, deleting, or updating a row. The following is a simple demonstration of just that:

```
--CONNECTION A
BEGIN TRANSACTION;
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SERIALIZABLE)
WHERE ArtistId >= 8;
```


WOW! eBook
www.wowebook.org

This returns one row for 'Starr'. Add the one row on CONNECTION B:

```
--CONNECTION B
INSERT INTO Art_InMem.Artist(ArtistId, Name)
VALUES (9,'Groening'); --See, brought him back!
```

Then executing the following:

```
--CONNECTION A
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId >= 8;
COMMIT;
```

The result set will return the same one row, but the second call will fail:

```
Msg 41325, Level 16, State 0, Line 803
The current transaction failed to commit due to a serializable validation failure.
```

When we created the table, we did not put a key on the Name column. What if this is a key column that you access single rows for, particularly for editing? Would this have an effect? To see, let's fetch a row by name on CONNECTION A:

```
--CONNECTION A
BEGIN TRANSACTION;
SELECT ArtistId, Name
FROM  Art_InMem.Artist WITH (SERIALIZABLE)
WHERE Name = 'Starr';
```

On a different connection, let's update a different row by name:

```
--CONNECTION B
UPDATE Art_InMem.Artist WITH (SNAPSHOT) --default to snapshot, but the change itself
                                        --behaves the same in any isolation level
SET    Padding = REPLICATE('a',4000) --just make a change
WHERE  Name = 'McCartney';
```

Going back to CONNECTION A, if we try to commit

```
--CONNECTION A
COMMIT;
```

we see an error:

```
Msg 41305, Level 16, State 0, Line 825
The current transaction failed to commit due to a repeatable read validation failure.
```


WOW! eBook
www.wowebook.org

Note that the error is a repeatable read validation failure, not serializable, since we updated a row. The message is a clue to what has gone wrong. The big point, though, is that just like on-disk tables, it is the rows it could touch that are serialized, not the rows that are actually returned. This is because an update could change the number of rows. If you are using a column as a key, it will behoove you to add a key.

```
ALTER TABLE Art_InMem.Artist
  ADD CONSTRAINT AKArtist UNIQUE NONCLUSTERED (Name) --A string column may be used to
                                                     --do ordered scans,
                                                     --particularly one like name
```

Repeat the experiment again and you will see no error on the commit. While you won't see any blocking when using in-memory tables, you could see tons of issues that behave like deadlocks if you are not careful. And while deadlocks are not horrible for data integrity, they are not great for performance.

Write Contention

In the previous sections, we looked at how readers and writers were isolated. The pattern was: read-only connections would never be blocked, would always see a consistent view of the database, and would fail at `COMMIT` if need be. In this section, we will now look at a less obvious set of cases, that of what happens when a connection that is writing to the table (including updates and deletes) contends with another writing connection.

In the following examples, I will show several scenarios that will occasionally come up, such as:

- Two users update the same row
- Two users delete the same row
- Two users insert a row with uniqueness collision
- A new row is inserted that matches a row that is being deleted on another connection

Not that these are all scenarios that can occur, but there are some of the more interesting things that we will see. First, two users try to modify the same row. So we update the same row on `CONNECTION A` and `CONNECTION B`:

```
--CONNECTION A
BEGIN TRANSACTION;
UPDATE Art_InMem.Artist WITH (SNAPSHOT)
SET    Padding = REPLICATE('a',4000) --just make a change
WHERE  Name = 'McCartney';
```

Then as soon as we execute the same on `CONNECTION B` (in or out of an explicit transaction):

```
--CONNECTION B
BEGIN TRANSACTION;
UPDATE Art_InMem.Artist WITH (SNAPSHOT)
SET    Padding = REPLICATE('a',4000) --just make a change
WHERE  Name = 'McCartney';
```


WOW! eBook
www.wowebook.org

we are greeted with

```
Msg 41302, Level 16, State 110, Line 3
The current transaction attempted to update a record that has been updated since this
transaction started. The transaction was aborted.
Msg 3998, Level 16, State 1, Line 1
Uncommittable transaction is detected at the end of the batch. The transaction is rolled
back.
The statement has been terminated.
```

Only one "dirty" version is allowed, so we get the immediate failure. Turns out that any combination of deletes and updates of the same physical resource will cause this to occur. Remember that a delete in in-memory OLTP is not a removal of the resource immediately, but rather an update to the ending timestamp.

Inserts are also interesting. On two connections, we will execute the following. We will use different manually created surrogate keys, because if we were using an identity key, we would get two different values because of the autonomous transaction.

```
--CONNECTION A
ROLLBACK TRANSACTION --from previous example
BEGIN TRANSACTION
INSERT INTO Art_InMem.Artist (ArtistId, Name)
VALUES  (11,'Wright');

--CONNECTION B
BEGIN TRANSACTION;
INSERT INTO Art_InMem.Artist (ArtistId, Name)
VALUES  (12,'Wright');
```

Both of these actually succeed. Look at the data on either connection and you will see a unique view of the data. So if two connections spin up and start inserting the same data in a large transaction, we could make a lot of new data, only to have the first connection succeed:

```
--CONNECTION A
COMMIT;
```

and the second one:

```
--CONNECTION B
COMMIT;
```

will give us a serializable failure (not a `UNIQUE` constraint collision, unfortunately). You will need to try that operation again to see that failure.

The final case for this section is if one connection deletes a row, but hasn't committed, what happens if an `INSERT` tries to reinsert that row? This one goes against the grain, actually.

```
--CONNECTION A
BEGIN TRANSACTION;
DELETE FROM Art_InMem.Artist WITH (SNAPSHOT)
WHERE ArtistId = 4;
```


WOW! eBook
www.wowebook.org

Then execution on a different connection, and insert with the same data:

```
--CONNECTION B --in or out of transaction
INSERT INTO Art_InMem.Artist (ArtistId, Name)
VALUES (4,'Picasso');
```

This will result in a uniqueness violation, not a "wait, someone is already working in this space" error.

```
Msg 2627, Level 14, State 1, Line 2
Violation of UNIQUE KEY constraint 'AKArtist'. Cannot insert duplicate key in object
'Artist'. The duplicate key value is (Picasso).
```

Finally, ROLLBACK the transaction so we can move on to the next code:

```
--CONNECTION A
ROLLBACK; --We like Picasso
```

As you can certainly see, the in-memory OLTP implementation of concurrency is vastly different from the version we saw for the on-disk structures. However, the implementation does one thing completely different and much better than the on-disk implementation. Rather than blocking or being blocked by the other sessions, the in-memory OLTP engine validates data consistency at transaction commit time, throwing an exception and rolling back the transaction if rules are violated. This does, however, vastly reduce the possibility that you will stomp over changes and not realize it.

Foreign Keys

This section covers one last consideration of using the memory-optimized model of concurrency. You saw when dealing with on-disk structures that unless you were using CASCADE operations, no locks were held on the related table when you made changes. This is not the case for in-memory tables. When you insert a row and it checks the existence of the parent row (or vice versa), a REPEATABLE READ contract is started. So if any part of the row is changed, you will get a failure.

For example, let's try to create a piece of artwork:

```
--CONNECTION A
BEGIN TRANSACTION
INSERT INTO Art_InMem.Artwork(ArtworkId, ArtistId, Name)
VALUES (5,4,'The Old Guitarist');
```

While we are out getting something to drink, another user changes a column on this artist row:

```
--CONNECTION B
UPDATE Art_InMem.Artist WITH (SNAPSHOT)
SET    Padding = REPLICATE('a',4000) --just make a change
WHERE ArtistId = 4;
```

Now we come back and COMMIT the transaction:

```
--CONNECTION A
COMMIT;
```


WOW! eBook
www.wowebook.org

We get the following error:

```
Msg 41305, Level 16, State 0, Line 82
The current transaction failed to commit due to a repeatable read validation failure.
```

SQL Server 2014 came out without foreign keys, because Microsoft didn't think that people who would employ such a feature made for high performance would use them. I somewhat agree, and this particular quirk in the version we have currently could make them less palatable. However, to implement your own foreign keys in code, you will need to either say "no deletes" or use `REPEATABLE READ` isolation level. I suspect that many who need blazing performance will opt for the no delete policy, which if you need to cross containers and use on-disk structures is the policy that you will use anyhow.

Coding for Asynchronous Contention

One of the most important considerations we need to make is how our software deals with the effects of delays in execution. So far in this chapter, I've discussed at length the different mechanisms, such as transactions, isolation levels, and so on, that SQL Server uses to protect one connection from another at a physical level. Once your connections are not internally using the same resources, it is up to you as a programmer to make sure that the state of the data is the same when it executes as when you actually apply some changes. The time between this caching of rows could be microseconds in the case of two `UPDATE` statements that were blocking one another, or data may have been purposefully cached (either due to blocking, isolation collisions, or a caching layer, or just a user who opened a form, made some changes, and didn't click "save" until after they finished their pepperoni Hot Pocket).

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, and make changes to the data or make some decision based on the retrieved values. Once the users have performed their operations, they may make save some additional database changes based on what they have seen.

For example, consider a customer row in a call center. It is a low probability that two people with access to the same account is calling into your sales call center on two lines talking to two different agents, making changes to the same account. But that low probability is not zero probability that two changes will be made to the same row simultaneously. Another example is an inventory level. You check to see how many widgets are available, then offer to sell one to the customer. Even if there were 1,000 items there a moment before, you will still check as you save the order to ensure that is still the case.

The idea of optimistic change detection is basically the same as it is for SQL Server, except now we need to deal with asynchronous changes to data, not the synchronous ones that SQL Server handles. The problem for we programmers is that every synchronous contention issue turns into an asynchronous one. User A updates row R at the same time that User B updates row R, and one is turned away or blocked. Particularly when using locks, if User B was blocked, the person issuing the query would not ever know that the row had changed, or even felt the latency if it lasted 10 milliseconds. But the row may have changed, and the second update could blow away the changes.

There are generally four ways this asynchronous timing issues is handled:

- *Chaos*: Just let it happen. If two users modify the same row in the database, the last user wins. This is not generally 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. Such an approach is never what I suggest for controlling your data resource modifications, but so far, I have not been elected data emperor, just data architect.

WOW! eBook
www.wowebook.org

- *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, reject the operation, and have the interface refresh the data from the table, showing the user what was changed. When optimistic locking is implemented, this is by far the most common method used.
- *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 change detection purposes.
- *Reality/hybrid*: The reality of this situation is that if you are modifying thousands of rows, caching the row states, and then checking it may not always be possible. Most of the time this is a matter of performance, in that adding another join to the cache of data isn't reasonable. In a true OLTP database, most large updates are done as maintenance during down time, so this may not be an issue. Using MVCC can also alleviate these issues.

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 Change Detection

A row-based scheme is used to check every row as to 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 is fetch data, modify data in the cached copy, 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. If you have any large datatypes, it can be much slower also. Checking all columns is typical 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 row modified 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` *(previously known as* `timestamp`*) column*: In the previous method, you used a manually controlled value to manage the optimistic lock value. This method uses a 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.

WOW! eBook
www.wowebook.org

The next two sections cover adding the optimistic lock columns to your tables and then using them in your code.

Adding Validation Columns

In this section, we'll add a column to a table to support adding either the `datetime2` 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`. Here's the structure:

```
CREATE SCHEMA Hr;
GO
CREATE TABLE Hr.person
(
     PersonId int 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`. We'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), implement an `INSTEAD OF` trigger on the `UPDATE` of the `Hr.Person` table:

```
CREATE TRIGGER Hr.Person$InsteadOfUpdateTrigger
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
```


WOW! eBook
www.wowebook.org

```
      --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 blocks]
          --[modification blocks]
          --remember to update ALL columns when building instead of triggers
          UPDATE Hr.Person
          SET    FirstName = inserted.FirstName,
                 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 Person.PersonId = inserted.PersonId;
   END TRY
      BEGIN CATCH
              IF XACT_STATE() > 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','19691212');

SELECT *
FROM   Hr.Person;
```

Now, you can see that the data has been created:

```
PersonId    FirstName    MiddleName   LastName  DateOfBirth
----------- ------------ ------------ --------- -----------
1           Paige        O            Anxtent   1969-12-12

RowLastModifyTime       RowModifiedByUserIdentifier
----------------------- ----------------------------
2016-06-11 14:43:45.576 SomeUserName
```


WOW! eBook
www.wowebook.org

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:

```
RowLastModifyTime
-----------------------
2016-06-11 14:44:48.791
```

If you want to set the value on INSERT, or implement RowCreatedBy -Date 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 UPDATE (typically I will add an INSTEAD OF INSERT TRIGGER, but will not for brevity, and I trust the programmer in this case to not set funky times for the 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 (note that the rowversion datatype is not supported in in-memory, and as I show in the downloads for Chapter 6, triggers are possible if tricky for in-memory change detection columns), and I almost always use a rowversion when implementing an optimistic mechanism. I usually have the row modification time and user columns on tables 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 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 rowversion 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, but while timestamp has been on the deprecation list for years, there is probably little chance of it going away.

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;
```


WOW! eBook
www.wowebook.org

You can see now that the rowversion has been added and magically set:

```
PersonId    RowVersion
----------- ------------------
1           0x00000000000007D1
```

Now, when the row is 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           0x00000000000007D2
```

■ **Caution** The rowversion datatype is an ever increasing number, so it can be useful for determining changes in the database after a particular rowversion value.

Coding for Row-Level Change Detection

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, even when changing the key value if allowed
  --non-key columns
  and   FirstName = 'Paige'
  and   MiddleName = 'ona'
  and   LastName = 'Anxtent'
  and   DateOfBirth = '19691212';
```


WOW! eBook
www.wowebook.org

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 = '2016-06-11 14:52:50.154';
```

Use a `rowversion` column:

```
UPDATE  Hr.Person
SET     FirstName = 'Fred'
WHERE   PersonId = 1
  and   RowVersion = 0x00000000000007D4;
```

Which is better performance-wise? Either of these generally performs just as well as the other (unless you have very large columns in the first case!), 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.

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
WHERE  PersonId = 1
  And  Rowversion = 0x00000000000007D5;
```

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 checking for changes because it is far simpler and safer and cannot be overridden by any code, no matter how you implement the other columns.

Coding for Logical Unit of Work Change Detection

Although row-based optimistic change checks 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 code always fetch not only the invoice line items but at least the invoice's timestamp into the client's

WOW! eBook
www.wowebook.org

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 IDENTITY(1,1),
     Number varchar(20) NOT NULL,
     ObjectVersion rowversion not null,
     CONSTRAINT PKInvoice PRIMARY KEY (InvoiceId)
);
--also ignoring 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 PKInvoiceLineItem primary key (InvoiceLineItemId),
     CONSTRAINT FKInvoiceLineItem$references$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:

```
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;
```


WOW! eBook
www.wowebook.org

```
    BEGIN TRY
        BEGIN TRANSACTION;
        SAVE TRANSACTION @savepoint;

        --tweak the ObjectVersion on the Invoice Table
        UPDATE  Invoicing.Invoice
        SET     Number = Number
        WHERE   InvoiceId = @InvoiceId
          And   ObjectVersion = @ObjectVersion;

        IF @@ROWCOUNT = 0
           THROW 50000,'The InvoiceId no longer exists or has been changed',1;

        DELETE  Invoicing.InvoiceLineItem
        FROM    InvoiceLineItem
        WHERE   InvoiceLineItemId = @InvoiceLineItemId;

        COMMIT TRANSACTION;

    END TRY
    BEGIN CATCH

        --if the tran is doomed, and the entryTrancount was 0,
        --we can 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,1;
        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.

WOW! eBook
www.wowebook.org

■ **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 is 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 little 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.
- *Use some form of optimistic mechanism in your code*: Use some form of optimistic mechanism to catch when rows have changed between the time you cached it and the actual operation happens, preferably using a `rowversion` column, because it requires the smallest amount of coding and is managed entirely by SQL Server. Whether the difference in time is a millisecond or a day, it is important to make sure you are updating the row you expected.
- *Consider using some form of the* `SNAPSHOT` *isolation level*: For on-disk structures, 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. All in-memory OLTP table access is in a flavor of snapshot. 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.

WOW! eBook
www.wowebook.org

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, and have adequate resources on your machine. As of SQL Server 2014 it can be far more complicated if you decide to use the new in-memory OLTP feature.

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.

WOW! eBook
www.wowebook.org

CHAPTER 12

Reusable Standard Database Components

The purpose—where I start—is the idea of use. It is not recycling, it's reuse.

—Issey Miyake, Japanese fashion designer and fragrance connoisseur

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), 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. In Chapter 9, I 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 complete 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. However, most projects will follow a common pattern that will look like something that has been done before. It isn't plagiarism to start from some initial design, and it definitely isn't plagiarism to use common objects that have been used by thousands of other people. If you are starting a very large project, you may want to look at previous models or perhaps even prebuilt "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 longer title than the book you hold in your hands right now). Karen Lopez (`@datachick` on Twitter) frequently speaks on the subject of universal models in the PASS universe that I am often 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 a few database constructs that I find to be useful and almost always the same for every database I create (where I have complete control, obviously!). 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 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 them mathematics-related).

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_12

WOW! eBook
www.wowebook.org

- *Calendar table*: A table where every row represents a day, assisting in grouping data for queries, both for reports and operational usage.
- *Utility objects*: Utilities to monitor usage of the system; extended DDL to support operations that aren't part of the base DDL in T-SQL. Every programmer has utility objects that they use to make their job easier.
- *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*: 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 by following the patterns of implementation I have discussed throughout the book thus far and the practices I will discuss in this chapter, you can produce databases that are sufficiently similar to enable the people supporting your work to easily figure out what you had in mind.

If you are dealing with a third-party system that forbids adding 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, but if you cannot follow that approach, another common approach is to create a companion database where you locate code you need to access their code from the database tier. You would need to slightly rework some examples presented in this chapter to use that approach, but that rework would be minimal.

For the examples in this chapter, I am going to use a copy of the new SQL Server 2016 `WideWorldImporters` database (which we have used in previous chapters already) to stick to the supposition of the chapter that you should place the tables in the database with the data you are working with. If you are working with a community version of `WideWorldImporters` that you cannot modify, you can build your own companion database for the examples. The examples are easily ported back to `AdventureWorks` if you are using it with an earlier version of SQL Server as well. I will include a comment in queries 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.

■ **Note** The code in this chapter will all be presented using on-disk tables. The objects from the first two sections will have in-memory OLTP versions in the download.

Numbers Table

A numbers table is a precalculated table of some number sequence, with the most typical being non-negative integers. The name "numbers" is pretty open ended, but getting so specific as `nonNegativeIntegers` is going to subject you to ridicule 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 in SQL Server 2012, 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.

WOW! eBook
www.wowebook.org

While you can make the numbers table contain any 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 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 numbers 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)

In the examples in this section, we will look at several techniques you may find useful, and possibly quite often. The following 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 (not to mention if you want to add attributes that are not easily calculated).

```
;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 (with 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. Breaking the code down, you get the following:

WOW! eBook
www.wowebook.org

```
;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))
```

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. (If you added `D3`, it would be `100*D3.I`.) 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
```


WOW! eBook
www.wowebook.org

```
3           90          93
4           90          94
5           90          95
6           90          96
7           90          97
8           90          98
9           90          99
```

This kind of combination of sets is a very useful technique in relational coding. 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` (`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 `Number` table is to make sure you get all values in a sequence, without having to loop value by value. Create this table in a schema named `Tools` to contain the types of tool objects, functions, and procedures you 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 WideWorldImporters;
GO
CREATE SCHEMA Tools;
GO
CREATE TABLE Tools.Number
(
    I   int CONSTRAINT PKNumber PRIMARY KEY
);
```

Then 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?

WOW! eBook
www.wowebook.org

```
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 just 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?). There is an example in the downloads of an esoteric example of what you can do with a table of numbers to do some pretty (nerdy) fun stuff, but for OLTP use, the goal will be (as we discussed in Chapter 5 on normalization) to precalculate values *only* when they are used often and can never change. Numbers-type tables are an excellent candidate for storing precalculated values because the sets 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*: In the downloads, there will be a supplemental file where 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 could write some looping code, or 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:

WOW! eBook
www.wowebook.org

```
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
```

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 so to 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 `WideWorldImporters` database that can provide us with an easy example set:

```
SELECT People.FullName, Number.I AS position,
             SUBSTRING(People.FullName,Number.I,1) AS [char],
             UNICODE(SUBSTRING(People.FullName, Number.I,1)) AS [Unicode]
FROM   /*WideWorldImporters.*/ Application.People
         JOIN Tools.Number
               ON Number.I <= LEN(People.FullName )
                   AND  UNICODE(SUBSTRING(People.FullName, Number.I,1)) IS NOT NULL
ORDER  BY FullName;
```

This returns 15128 rows (one for each character in a full name) in < 1 second on a virtual machine hosted on my writing tablet (which admittedly is a beefy tablet: Surface Pro 4; 16GB; 256GB SSD):

```
FullName              position    char Unicode
--------------------- ----------- ---- -----------
Aahlada Thota         1           A    65
Aahlada Thota         2           a    97
Aahlada Thota         3           h    104
Aahlada Thota         4           l    108
Aahlada Thota         5           a    97
Aahlada Thota         6           d    100
Aahlada Thota         7           a    97
Aahlada Thota         8                32
.......               ...         .    ...
```


WOW! eBook
www.wowebook.org

```
Aakarsha Nookala        1           A    65
Aakarsha Nookala        2           a    97
Aakarsha Nookala        3           k    107
Aakarsha Nookala        4           a    97
Aakarsha Nookala        5           r    114
Aakarsha Nookala        6           s    115
Aakarsha Nookala        7           h    104
Aakarsha Nookala        8           a    97
.......                 ...         .    ...
```

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  People.FullName, Number.I AS Position, return
        SUBSTRING(People.FullName,Number.I,1) AS [Char],
             UNICODE(SUBSTRING(People.FullName, Number.I,1)) AS [Unicode]
FROM   /*WideWorldImporters.*/ Application.People
         JOIN Tools.Number
               ON Number.I <= LEN(People.FullName )
                   AND  UNICODE(SUBSTRING(People.FullName, Number.I,1)) IS NOT NULL
WHERE  SUBSTRING(People.FullName, Number.I,1) NOT LIKE '[a-zA-Z ~''~-]' ESCAPE '~'
ORDER  BY FullName;
```

This returns the following:

```
FullName                                            Position    Char Unicode
--------------------------------------------------- ----------- ---- -----------
Abhoy PrabhupÄda                                   15              129
BahadÄ±r Korkmaz                                    7          ±   177
Bimla PrabhupÄda                                   15              129
Deviprasad PrabhupÄda                              20              129
Himadri PrabhupÄda                                 17              129
Ivica LuÄic                                        10              141
Malay PrabhupÄda                                   15              129
Sevim AydÄ±n                                       11          ±   177
Taner YÄ±lmaz                                       9          ±   177
Tereza PinÄakova                                   12              143
VÄ›ra Kopecka                                       3          ›   8250
VÄ›ra Stejskalova                                   3          ›   8250
Vicente ChÃ¡vez                                    12          ¡   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

A common issue when using a column that is supposed to have sequential values is that there can be gaps in values. For most cases, like a surrogate key, 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 (perhaps to figure out when you lost

WOW! eBook
www.wowebook.org

some identity values to errors). 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 `WideWorldImporters` database. Running the following query, you can see that not every `PersonId` is used:

```
SELECT  MIN(PersonId) AS MinValue, MAX(PersonId) AS MaxValue,
        MAX(PersonId) - MIN(PersonId) + 1 AS ExpectedNumberOfRows,
        COUNT(*) AS NumberOfRows,
        MAX(PersonId) - COUNT(*) AS MissingRows
FROM    /*WideWorldImporters.*/ Application.People;
```

This returns the following:

```
MinValue    MaxValue    ExpectedNumberOfRows NumberOfRows MissingRows
----------- ----------- -------------------- ------------ -----------
1           3261        3261                 1111         2150
```

So you know that there are 1111 rows "missing" between `BusinessEntityID` values 1 and 3261 (not that there is anything wrong with that in this case, not at all). To discover these rows, take a set of values from 1 to 3261 with no gaps, and subtract the rows using the `EXCEPT` relational operator:

```
SELECT Number.I
FROM   Tools.Number
WHERE  I BETWEEN 1 AND 3261
EXCEPT
SELECT PersonId
FROM    /* WideWorldImporters.*/ Application.People;
```

Execute this query and you will find that 2150 rows are returned, the 2150 surrogate key values that are missing. If this table needed sequential numbers (like if this was not a surrogate key value), you would have the values you needed to fill in. 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.

Separating Comma-Delimited Items

My last example that you can translate to a direct business need (particularly if you are not yet using SQL Server 2016) 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).

For our example, consider the string `'1,2,3'`. Sometimes we need to break this apart (perhaps for a parameter or, as we will see later in the section, to fix some normalization issue). In SQL Server 2016, there is a new `STRING_SPLIT` function, so you can say:

```
SELECT *
FROM   STRING_SPLIT('1,2,3',',');
```


WOW! eBook
www.wowebook.org

and the output will be:

```
value
--------------------
1
2
3
```

As wonderful as this is (and it is pretty wonderful), the following technique may still find its way into your code at times, especially if you cannot upgrade to SQL Server 2016 even before the next version of this book comes out!

```
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 output remains the same. The way this code works is pretty interesting in and of itself, and understanding it can help your understanding of SQL. 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 line limits the rows in `Tools.Number` to more than the length of the `@delimitedList` variable. The third line 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:

```
I
--------------------
1
3
5
```


WOW! eBook
www.wowebook.org

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 values up to the next comma. This sort of use of the numbers table will allow you to do what at first seems like it would require a painful, iterating algorithm in order to touch each position in the string individually (which is fairly slow in T-SQL, and a bit less so in CLR code) and does it all in one statement 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
```


WOW! eBook
www.wowebook.org

Ah…that's much better. Using STRING_SPLIT() we can do it this way:

```
SELECT poorDesign.poorDesignId AS betterDesignId, stringSplit.value AS betterScalarValue
FROM   dbo.poorDesign
          CROSS APPLY STRING_SPLIT(badValue,',') AS stringSplit;
```

In both outputs, 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 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 badly designed table.

One thing that is good about having both methods is that if you have a real scenario with millions of rows of this dreck, you could try both. The numbers table version might allow you more leeway to do more tuning, and perhaps complex splitting in any case.

I won't create the better design, but we do need to clean up the poorDesign table with the following, lest someone stumble upon it and use it as a good idea:

```
DROP TABLE dbo.poorDesign;
```

Calendar Table

A common task that people want to know how to do is perform 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, or at least pretty messy or impossible if you want something more than a simple calendar year and month grouping. 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 show you how to add later in the section (again, I am working in a copy of WideWorldImporters, 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
);
```


WOW! eBook
www.wowebook.org

■ **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 nondeterministic), which allows for specification of regionalization information. I didn't change my example to use `FORMAT` because `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. A stretch perhaps, but naming is pretty difficult at times, especially when working in a namespace in which 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 '20200101' --set the date range
ORDER   BY DateValue;
```


WOW! eBook
www.wowebook.org

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 the `Sales.Order` table in the `WideWorldImporters` database. This is why there is a `Year` column in the table:

```
SELECT Calendar.Year, COUNT(*) AS OrderCount
FROM   /*WideWorldImporters.*/ Sales.Orders
         JOIN Tools.Calendar
               --note, the cast here could be a real performance killer
               --consider using date columns where possible
            ON Orders.OrderDate = Calendar.DateValue --OrderDate is a date type column
GROUP BY Calendar.Year
ORDER BY Calendar.Year;
```

This returns the following:

```
Year             OrderCount
---------------- -----------
2013             19450
2014             21199
2015             23329
2016             9617
```

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 a natural relational coding style/technique. For example, to count the sales on Tuesdays and Thursdays:

```
SELECT Calendar.DayName, COUNT(*) as OrderCount
FROM   /*WideWorldImporters.*/ Sales.Orders
         JOIN Tools.Calendar
               --note, the cast here could be a real performance killer
               --consider using date columns where possible
            ON CAST(Orders.OrderDate as date) = Calendar.DateValue
WHERE DayName IN ('Tuesday','Thursday')
GROUP BY Calendar.DayName
ORDER BY Calendar.DayName;
```

This returns the following:

```
DayName    OrderCount
---------- -----------
Thursday   13421
Tuesday    13737
```

■ **Tip** In many tables where the value is a `datetime` or `datetime2` and I need to use just the date often, I will add a computed column that has the date only.

WOW! eBook
www.wowebook.org

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
(
    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 = 2016
  AND  rowNbr = 1;
```

This returns the following set of values that appears to be the result of some advanced calculations, but rather is the results of a fairly simple query (to anyone in Chapter 12 of this book, for sure!):

```
Year              MonthName  Day
----------------- ---------- ----
2016              January    19
2016              February   16
2016              March      15
2016              April      19
2016              May        17
2016              June       14
2016              July       19
2016              August     16
2016              September  20
2016              October    18
2016              November   15
2016              December   20
```


WOW! eBook
www.wowebook.org

Now, utilizing another CTE, you could use this to join to the `Sales.Order` 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 the mad hatter in charge of marketing (I think they prefer the term *marketing analyst*) and would be done in no time. However, if this date became a corporate standard-desired date (and no other jobs were available for you), 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 usually given day every month, so the report needs to know how sales were for four days after. So, perhaps the column would be `BigSaleDayFlag`, 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.

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 is to add a relative positioning value to the table so that a user 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 added the following columns that I load with increasing values with no gaps:

```
RelativeDayCount int NOT NULL,
RelativeWeekCount int NOT NULL,
RelativeMonthCount int NOT NULL
```

Using these columns, I can 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:

WOW! eBook
www.wowebook.org

```
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,

        --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
```


WOW! eBook
www.wowebook.org

```
        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
        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 '20200101'; --set the date range
```

Now I 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;
```


WOW! eBook
www.wowebook.org

This returns the following:

```
FiscalYear OrderCount
---------- -----------
2013       950
2014       1723
2015       2109
2016       1782
```

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 = '20140509';

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
---------------- ---------- -----------------
2014-04-20       2014-05-09 747
2014-04-21       2014-05-09 747
2014-04-22       2014-05-09 747
2014-04-23       2014-05-09 747
2014-04-24       2014-05-09 747
2014-04-25       2014-05-09 747
2014-04-26       2014-05-09 747
2014-04-27       2014-05-09 748
2014-04-28       2014-05-09 748
2014-04-29       2014-05-09 748
2014-04-30       2014-05-09 748
2014-05-01       2014-05-09 748
2014-05-02       2014-05-09 748
2014-05-03       2014-05-09 748
```

From this, we can see that the previous two weeks start on Sunday (04/20/2014 and 04/27/2014) and end on Saturday (04/26/2014 and 05/03/2014). The dates 05/04/2012 to 05/09/2012 are not included because that is this current 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:

WOW! eBook
www.wowebook.org

```
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. 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 = '20140509';

SELECT MIN(Calendar.DateValue) AS MinDate, MAX(Calendar.DateValue) AS MaxDate
FROM   Tools.Calendar
           JOIN (SELECT *
                 FROM Tools.Calendar
                 WHERE DateValue = @interestingDate) AS CurrentDate
              ON  Calendar.RelativeMonthCount < (CurrentDate.RelativeMonthCount)
                  AND Calendar.RelativeMonthCount >=
                                    (CurrentDate.RelativeMonthCount -12);
```

This returns the following:

```
MinDate    MaxDate
---------- ----------
2013-05-01 2014-04-30
```

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 `WorldWideImporters` database around September 27, 2008. (Keep in mind that the sample databases could change in the future to get more up-to-date data.)

```
DECLARE @interestingDate date = '20140509'

SELECT Calendar.Year, Calendar.Month, COUNT(*) AS OrderCount
FROM   /*WorldWideImporters.*/ Sales.Orders
         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 Orders.ExpectedDeliveryDate = Calendar.DateValue
GROUP BY Calendar.Year, Calendar.Month
ORDER BY Calendar.Year, Calendar.Month;
```

This query will give you items that shipped in the previous ten months and in May, and group them by month, as follows:

WOW! eBook
www.wowebook.org

```
Year            Month  OrderCount
--------------- ------ -----------
2013            7      1969
2013            8      1502
2013            9      1664
2013            10     1617
2013            11     1531
2013            12     1545
2014            1      1774
2014            2      1595
2014            3      1561
2014            4      1742
2014            5      1867
```

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 you can easily create more fiscal calendars, reporting calendars, corporate holiday calendars, sales calendars, and so forth by adding more columns to this very same calendar table. For instance, you may have multiple business units, each with its 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), 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.

Some tools, however, go beyond the type of utilities that are used infrequently. 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 20th 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 equally 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 we would have abnormal procedures spread around the enterprise with names that included a database 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 that was managed on every server. Ideally, they were identical, but over time, the DBAs and or developers used the database for "temp" usages that again became unwieldy and decidedly not temporary.

WOW! eBook
www.wowebook.org

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 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, but not multiple tables and such.

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), by 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. In my real systems we have this sort of object, as well as many based on the dynamic management views (DMVs) that capture statistics about the database daily, or sometimes hourly.

For monitoring data, I will create a `Monitor` schema. To this I will not give rights to anyone other than the `db_owner` users by default, as the `Monitor` schema will generally be for the DBA to get a feel for system growth, not for general usage.

```
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 base structures.

```
CREATE TABLE Monitor.TableRowCount
(
        SchemaName  sysname NOT NULL,
        TableName   sysname NOT NULL,
        CaptureDate date    NOT NULL,
        Rows        int NOT NULL, --proper name, rowcount is reserved
        ObjectType  sysname NOT NULL,
        Constraint PKTableRowCount PRIMARY KEY (SchemaName, TableName, CaptureDate)
);
```


WOW! eBook
www.wowebook.org

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
--
-- 2016 Louis Davidson - drsql@hotmail.com - drsql.org
-- ------------------------------------------------------------------

-- The CTE is used to set up the 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 WideWorldImporters, where we have been working:

WOW! eBook
www.wowebook.org

```
EXEC Monitor.TableRowCount$CaptureRowcounts;

SELECT *
FROM   Monitor.TableRowCount
WHERE  SchemaName = 'Purchasing'
ORDER BY SchemaName, TableName;
```

Then (assuming you are still in the pristine version of the `WideWorldImporters` database we are working on), the output of this batch will be as follows:

```
SchemaName      TableName                    CaptureDate Rows        ObjectType
--------------- ---------------------------- ----------- ----------- --------------
Purchasing      PurchaseOrderLines           2016-06-15  8367        USER_TABLE
Purchasing      PurchaseOrders               2016-06-15  2074        USER_TABLE
Purchasing      SupplierCategories           2016-06-15  9           USER_TABLE
Purchasing      SupplierCategories_Archive   2016-06-15  1           USER_TABLE
Purchasing      Suppliers                    2016-06-15  13          USER_TABLE
Purchasing      Suppliers_Archive            2016-06-15  13          USER_TABLE
Purchasing      SupplierTransactions         2016-06-15  2438        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 each 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 = '%',
    @doFkFlag bit = 1,
    @doCkFlag bit = 1
) as
-- ------------------------------------------------------------------
-- Enables disabled foreign key and check constraints, and sets
```


WOW! eBook
www.wowebook.org

```
-- 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
--
-- 2016 Louis Davidson - drsql@hotmail.com - drsql.org
-- ------------------------------------------------------------------

 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 @doFkFlag = 1)
                    OR (constraintType = 'CHECK_CONSTRAINT' AND @doCkFlag = 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
```


WOW! eBook
www.wowebook.org

```
                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
                SELECT 'Error occurred: ' + CAST(ERROR_NUMBER() AS varchar(10))+ ':' +
                        error_message() + CHAR(13) + CHAR(10) + 'Statement executed: ' +
                        @statement;
            END CATCH
        END;

    END;
```

I have several more of these available in the downloads of my web site (www.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 9 we implemented an audit trail using the audit feature. In this section, the types of logging we will want to do are a more generic form of logging that is intended 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 `ErrorHandling.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 SCHEMA ErrorHandling;
GO
CREATE TABLE ErrorHandling.ErrorLog(
        ErrorLogId int NOT NULL IDENTITY CONSTRAINT PKErrorLog PRIMARY KEY,
                Number int NOT NULL,
        Location sysname NOT NULL,
```


WOW! eBook
www.wowebook.org

```
        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 ErrorHandling.ErrorLog$Insert
(
        @ERROR_NUMBER int,
        @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
--
-- 2016 Louis Davidson - drsql@hotmail.com - drsql.org
-- ------------------------------------------------------------------------------------

 BEGIN
        SET NOCOUNT ON;
        BEGIN TRY
           INSERT INTO ErrorHandling.ErrorLog(Number, Location,Message)
           SELECT @ERROR_NUMBER,COALESCE(@ERROR_LOCATION,'No Object'),@ERROR_MESSAGE;
        END TRY
        BEGIN CATCH
           INSERT INTO ErrorHandling.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',1;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0
        ROLLBACK TRANSACTION;

    --[Error logging section]
        DECLARE @ERROR_NUMBER int = ERROR_NUMBER(),
                @ERROR_PROCEDURE sysname = ERROR_PROCEDURE(),
```


WOW! eBook
www.wowebook.org

```
                @ERROR_MESSAGE varchar(4000) = ERROR_MESSAGE();
        EXEC ErrorHandling.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 1, Line 3
Test error
```

And checking the ErrorLog:

```
SELECT *
FROM  ErrorHandling.ErrorLog;
```

we can see that the error is logged:

```
ErrorLogId  Number  Location    Message       LogTime                    ServerPrincipal
----------- ------- ----------- ------------- -------------------------- ------------------
1           50000   No Object   Test error    2016-06-15 15:31:52.649    SomeUserName
```

This basic error-logging procedure can make it much easier to understand what has gone wrong when a user has an error (or see when hundreds of errors have occurred, and the code has just TRY...CATCH hidden it from the user). 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 and are not building reasonably simple error-handler stacks, in which all the ErrorHandling.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.

WOW! eBook
www.wowebook.org

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 "Separating Comma-Delimited Items." We might create a function `Tools.String$SplitOnCommaOrSemicolon(@ValueToSplit)` that encapsulates the string-splitting function for reuse that uses a couple of delimiters simultaneously. Another common example is changing a time from one time zone (often UTC) to another.
- *Security*: Microsoft provides a lot of security-oriented system functions for you to determine who the user is, but sometimes they aren't enough. For some systems, I have employed a function in a schema named `Security` to allow for overriding the system context information when needed. The goal is that whenever you need to know what user is making changes to your system, you can call the `Security.UserName$get()` function and know that it is doing the heavy lifting, either looking to the `ORIGINAL_LOGIN()` system function or `SESSION_CONTEXT`, or even 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.

As I 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

Most chapters in this book 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 and maintain a library of SQL that you can reuse. 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 an 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.

WOW! eBook
www.wowebook.org

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 implement methods of grouping date data using typical relational methods, both for normal date groupings like year, month, etc., and for custom times like corporate fiscal calendars, sales, holidays, etc. While the most common usage might be for data warehouse/reporting systems (which you will see in Chapter 14), having the calendar table available in your OLTP database can be very useful.
- *Utilities*: 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. Every programmer has utilities that they use to make their job easier. 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 now and should always be to make the database its own universe. With the contained database features in 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 AM about why something failed because another database wasn't available when another needed it. That can't be a bad thing, can it?

WOW! eBook
www.wowebook.org

CHAPTER 13

Architecting Your System

Men's arguments often prove nothing but their wishes.

Charles Caleb Colton, English cleric, writer, and collector

So far in the book, we started with how one should design a database without regard for technology, then covered how one might implement that database using SQL Server. So far the process has been one of building one precept upon another. The one topic we really haven't breached is data access. But as SQL Server grows in complexity, we are given more and more choices. It is essential for the architect who is creating a new system, or even changing an existing system, to think about the architectural building blocks and make sure that a number of things are matched up:

- *Technical needs*: What does your application really need? Amazing throughput? Massive data quantities? Extreme data integrity?
- *Human abilities*: My personal feeling is to never limit what tech you use to the abilities of the programmers, but not everyone can do everything. Plus, some methods of architecting the code layer for a system are more time consuming, no matter the programmer.
- *Tools*: What tools will the programmers who are building the front end be using? Very few data architects will smile happily when the three letters ORM are uttered, but object-relational mapping tools are popular.

In this chapter I will provide a rather brief overview of some of the architectural decisions you need to make, along with the pros and cons of each—in other words, "opinions" (technical-based opinions, but opinions nevertheless). Regardless of whether your application is a good, old-fashioned, client-server application, a multitier web application, uses an ORM, or uses some new application pattern that hasn't been created (or I haven't heard of!) as I write this, data must be stored in and retrieved from tables. Most of the advice presented in this chapter will be relevant regardless of the type of application you're building.

In this chapter, I will present a number of my own 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.

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_13

WOW! eBook
www.wowebook.org

In this chapter, I am going to discuss the following topics:

- *Choosing the engine for your needs*: I will give a brief overview of how the integration between the two engines works, and how to decide when to use the in-memory model.
- *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 the CLR in T-SQL*: 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. Again, bear in mind that a lot of this is just my personal opinion.

■ **Note** I previously mentioned object-relational mapping tools (examples of which include Hibernate, Spring, or even the ADO.NET Entity Framework), so you might be thinking that I'm going to cover them in this chapter. In the end, however, these tools are really using ad hoc access, in that they are generating SQL. 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 ORM 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 had 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 suchlike. 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.

WOW! eBook
www.wowebook.org

■ **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.

Choosing the Engine for Your Needs

Throughout the latter half of the book, you have seen some discussion of the in-memory OLTP tables, and how they are different (and how they are similar). However, we haven't talked too much about the overall architecture, sticking mostly to the ways the in-memory OLTP tables themselves have worked when used in a "normal" T-SQL statement. In this section I want to discuss the architecture a bit more, to help you get a feel for whether or not it is something you should use for systems you are building. In the rest of this chapter, I will make only mention of the differences when they are considerable the discussion of data access.

When we talk about the in-memory engine, it is basically a complete, complementary storage and query-processing system. Data is stored in the same database along with on-disk data, but in a completely different format. All data you want to store in these tables must fit within RAM, including (as discussed in Chapter 11) versions of rows that are being changed. Logging is different, in that your indexes are never logged, and you can have completely unlogged (not durable) tables. Microsoft has made it so that using the in-memory model is super easy, other than a few things:

- *You generally need at least double the RAM that you have data*: For version data, indexes, etc. Unless you have small data sets, this could be tough for a lot of organizations.
- *The concurrency model*: All concurrency is implemented using a snapshot mentality. After you start a transaction and it uses data in the in-memory container, you cannot see any other user's changes.
- *Intra-table/intra-row: data integrity is more difficult*: Unless you can do your data integrity checks using foreign keys, uniqueness constraints, it can be difficult to check data because of the concurrency model, particularly if lots of data is changing in a lot of tables.

Fortunately, the logical model of our database needn't change that much to use the in-memory model, and neither do our tables for the most part. Even better for a designer, other than a few data integrity concerns, there is no real difficulty with having a few tables in each engine. An example could be

- *Sales order tables in in-memory tables*: We need massive throughput in these tables, with hundreds of orders per second coming in.
- *Customer data in on-disk tables*: Data changes some, but many of the orders come from repeat customers. You could implement business rules that customers are not deleted (at least not using a simple `DELETE` statement, but perhaps as part of some cleanup processes that are carefully created). The customer identifiers are cached in the code when the customer logs in, and read contention is far less of a concern than write contention.

Interaction between the engines is seamless for the programmer. As we have seen throughout the book when accessing in-memory tables, we simply execute a normal SQL statement. This uses what is called query "interop" mode, letting typical interpreted T-SQL the data in in-memory structures.

WOW! eBook
www.wowebook.org

You can see large gains in performance simply by using interop mode, but where the engine truly shines is when you use natively compiled objects. While they are written using T-SQL (a heavily limited version of T-SQL, but it looks just like T-SQL with a few new settings), they are compiled into binary code like a typical C# program (just like the in-memory tables). This gives you all of the benefits of a compiled language coupled with the benefits of the relational engine.

My goal is to just give you a few guidelines on when to choose to use the in-memory engine. Microsoft provides its own list at `https://msdn.microsoft.com/en-us/library/dn133186.aspx`, and since this is such a new engine (the 2014 version was extremely limited, whereas the 2016 version has become far more usable), Microsoft may expand the expected use cases before I write a new edition of this book. My list of characteristics for potential utilization includes

- *High concurrency needs with a low chance of collision*: When you have the need to load, modify, etc. many rows constantly/continuously, but there is little chance that two connections will contend with another. The way concurrency collisions are resolved will be costlier than short-term blocks.
- *Limited need for transaction isolation/short transactions*: The longer the transactions your system needs to make, the more complicated things will get, not just due to collisions, but also because versions of rows are held so every connection gets their consistent view.
- *Limited search criteria that needs optimized*: With only 8 indexes per table, you likely can't have 20 different ways you want to search for data in an optimum manner.
- *Limited data integrity needs*: As all transactions are in a snapshot-based transaction, you can't see any external data until you commit (and the checks for serializable transactions are scans of table or indexes).

Basically speaking, the best candidates are the tables in a strict OLTP workload with the highest concurrent usage. But memory-optimized tables and natively compiled objects are not a way to fix poorly written code. If you have bad code, this could easily make it worse. Your applications probably need at least some change to ensure in-memory works well with your applications. As in all things, be sure and test out what works, and what works really well. We have covered many of the limitations and differences between the two engines, but there is more to read and learn than I have covered, particularly for management of these processes.

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` (as well as any other) statements textually 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). You can also compile your ad hoc SQL into a temporary plan.

I will make no distinction between ad hoc calls that are generated manually and those that use a middleware setup like any ORM: 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.)

WOW! eBook
www.wowebook.org

Sending queries as strings of text is the way that plenty of tools tend to converse with SQL Server, and is, for example, how SQL Server Management Studio does its interaction with the server metadata. If you have never used Extended Events to watch the SQL that any of the management and development tools use, 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.

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 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 `Chapter13` that will allow in-memory tables by default.

```
CREATE SCHEMA Sales;
GO
CREATE TABLE Sales.Contact
(
    ContactId   int CONSTRAINT PKContact PRIMARY KEY,
    FirstName   varchar(30),
    LastName    varchar(30),
    CompanyName varchar(100),
```


WOW! eBook
www.wowebook.org

```
    SalesLevelId  int, --real table would implement as a foreign key
    ContactNotes  varchar(max),
    CONSTRAINT AKContact 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
```

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 user 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.

WOW! eBook
www.wowebook.org

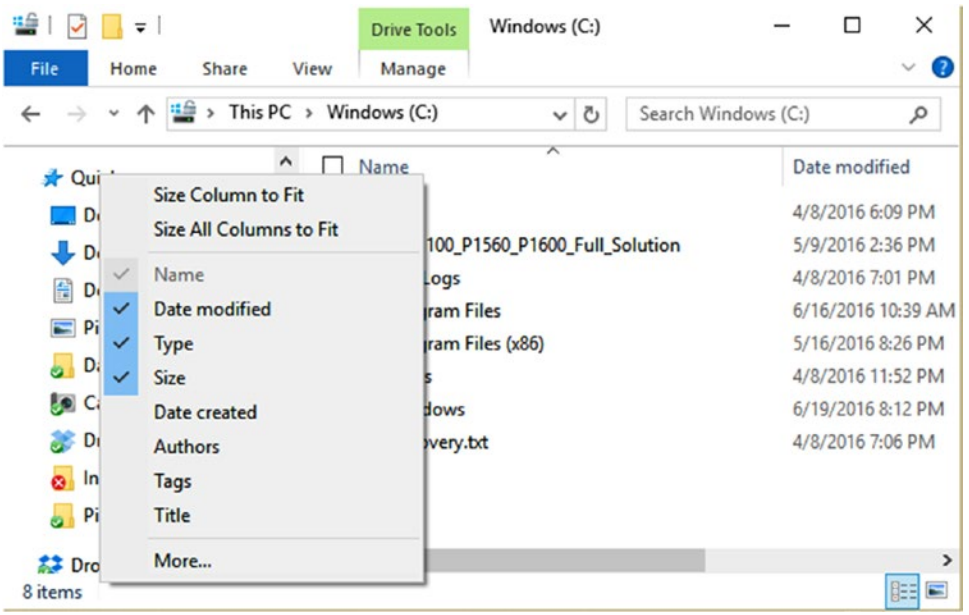


Figure 13-1. *The Windows file-listing dialog*

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 PKPurchase PRIMARY KEY,
    Amount      numeric(10,2),
    PurchaseDate date,
    ContactId   int
        CONSTRAINT FKContact$hasPurchasesIn$Sales_Purchase
            REFERENCES Sales.Contact(ContactId)
);
INSERT INTO Sales.Purchase(PurchaseId, Amount, PurchaseDate, ContactId)
VALUES (1,100.00,'2016-05-12',1),(2,200.00,'2016-05-10',1),
       (3,100.00,'2016-05-12',2),(4,300.00,'2016-05-12',1),
       (5,100.00,'2016-04-11',1),(6,5500.00,'2016-05-14',2),
       (7,100.00,'2016-04-01',1),(8,1020.00,'2016-06-03',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.

WOW! eBook
www.wowebook.org

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%';
```

This returns the following:

```
ContactId   FirstName  LastName   YearToDateSales  LastSaleDate
----------- ---------- ---------- ---------------- ------------
2           Jon        Rettre     6620.00          2016-06-03
```

If the user doesn't ask for a sales summary column, the client will send only the code that is not commented out in the following 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 'Karr%';
```

This returns only the following:

```
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. Even if it isn't terribly costly, it is still a waste of resources.

WOW! eBook
www.wowebook.org

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 values 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 ContactNotes columns for the row with ContactId = 1 contain 300MB. 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 and LastName columns only (which probably has less overhead than changing the SalesLevelId value that minimally has a foreign key), you can simply execute the following code:

```
UPDATE Sales.Contact
SET    FirstName = 'John',
       LastName = 'Ritter'
WHERE  ContactId = 2;
```

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%';
```

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%';
```


WOW! eBook
www.wowebook.org

If you think this looks like a very silly query to execute, you are generally correct. If you were writing this query to be used repeatedly, 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 will also play to the downside of any dynamically built statement. 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 @FirstNameLike
          AND  LastName LIKE @LastNameLike;
ELSE
        SELECT FirstName, LastName, CompanyName
        FROM   Sales.Contact
        WHERE  FirstName LIKE @FirstNameLike;
```

This is generally a bad idea, as you ideally want to have every query in a procedure to execute. Even worse, you can do something messy like the following in your WHERE clause so if any value is passed in, it uses it, or uses '%' otherwise:

```
WHERE  FirstName LIKE  ISNULL(NULLIF(LTRIM (@FirstNameLike) +'%','%'),FirstName)
  AND LastName LIKE ISNULL(NULLIF(LTRIM(@LastNameLike) +'%','%'),LastName)
```

Unfortunately though, this often does not optimize very well (particularly with a lot of data) 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$Get @FirstNameLike, @LastNameLike;
ELSE
        EXECUTE Sales.Contact$GetByLastNameLike @LastNameLike;
```


WOW! eBook
www.wowebook.org

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 a UI can know which parameters are applicable, handling this situation using ad hoc SQL can be far easier. 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): I 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 many 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 WideWorldImporters tables for this example, as that database has a nice amount of data to work with.)

```
USE WideWorldImporters;
GO
SELECT People.FullName, Orders.OrderDate
FROM   Sales.Orders
                JOIN Application.People
                       ON Orders.ContactPersonID = People.PersonID
WHERE  People.FullName = N'Bala Dixit';
```

Next, run the following query. See whether you can spot the difference between the two queries.

```
SELECT People.FullName, Orders.OrderDate
FROM   Sales.Orders
                JOIN Application.People
                       on Orders.ContactPersonID = People.PersonID
WHERE  People.FullName = N'Bala Dixit';
```


WOW! eBook
www.wowebook.org

These queries can't share plans because ON in the first query's FROM clause is uppercase and on in the second query's FROM clause is lowercase. Using the sys.dm_exec_query_stats DMV, you can see that the case difference does cause two plans by running

```
SELECT  *
FROM    (SELECT qs.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 People.FullName, Orders.OrderDate%';
```

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, and if you messed up entering/copying the query like I did a few 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 People.FullName
FROM   Application.People
WHERE  People.FullName = N'Bala Dixit';
```

The plan (from using showplan_text on in the manner we introduced in Chapter 10 "Basic Index Usage Patterns" as follows):

```
  |--Index Seek(OBJECT:([WideWorldImporters].[Application].[People].
                                                  [IX_Application_People_FullName]),
                SEEK:([WideWorldImporters].[Application].[People].[FullName]
                                                 =[@1]) ORDERED FORWARD)
```

The value of N'Bala Dexit' 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
```


WOW! eBook
www.wowebook.org

```
FROM   Person.Address AS address
         JOIN Person.StateProvince AS state
                ON address.StateProvinceID = state.StateProvinceID
WHERE  address.AddressLine1 ='1, rue Pierre-Demoulin';
```

The plan won't recognize the literal and parameterize it:

```
|--Nested Loops(Inner Join,
       OUTER REFERENCES:([WideWorldImporters].[Sales].[Orders].[OrderID], [Expr1004]) WITH
                                                                    UNORDERED PREFETCH)
       |--Nested Loops(Inner Join,
                  OUTER REFERENCES:([WideWorldImporters].[Application].[People].[PersonID]))
       |    |--Index Seek(OBJECT:([WideWorldImporters].[Application].[People].
                                                    [IX_Application_People_FullName]),
                  SEEK:([WideWorldImporters].[Application].[People].[FullName]
                                                    =N'Bala Dixit') ORDERED FORWARD)
       |    |--Index Seek(OBJECT:([WideWorldImporters].[Sales].[Orders].
                                                   [FK_Sales_Orders_ContactPersonID]),
                  SEEK:([WideWorldImporters].[Sales].[Orders].[ContactPersonID]
                  =[WideWorldImporters].[Application].[People].[PersonID]) ORDERED FORWARD)
       |--Clustered Index Seek(OBJECT:
                  ([WideWorldImporters].[Sales].[Orders].[PK_Sales_Orders]),
                        SEEK:([WideWorldImporters].[Sales].[Orders].[OrderID]=
                     [WideWorldImporters].[Sales].[Orders].[OrderID])
                                                                 LOOKUP ORDERED FORWARD)
```

Clearly it's no longer a "simple" plan. Now the literal from the query is still in the plan as `N'Bala Dixit'`, 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'Bala Dixit'` 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 WideWorldImporters
    SET PARAMETERIZATION FORCED;
```

Try the plan of the query with the join. It now has replaced the `N'Bala Dixit'` with `[@0]`, and the query processor can reuse this plan no matter what the value for the literal is. Note that there is a reason that more than simple plans are not parameterized by default. 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.

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';
```


WOW! eBook
www.wowebook.org

The plan will contain the literal, rather than the parameter, because it cannot parameterize the second and third arguments of a LIKE operator comparison (the arguments of the LIKE operator are `arg1 LIKE arg2 [ESCAPE arg3]`; we are only using the first two in our example). The following is a partial plan:

```
WHERE:([WideWorldImporters].[Application].[People].[FullName] like N'Bala Dixit')
                                                                    ORDERED FORWARD)
```

If you change the query to end with `WHERE N'Bala Dixit' LIKE People.FullName`, 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, which was a very long time ago indeed). Performance when using ad hoc calls to the SQL Server in your applications can certainly be less of a worry than you might have been led to believe. However, don't stop reading here. While performance may not suffer tremendously, the inability to easily performance tune queries 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 @FullName nvarchar(60) = N'Bala Dixit',
           @Query nvarchar(500),
        @Parameters nvarchar(500)

SET @Query= N'SELECT People.FullName, Orders.OrderDate
                        FROM   Sales.Orders
                                       JOIN Application.People
                                              on Orders.ContactPersonID = People.PersonID
                        WHERE  People.FullName LIKE @FullName';
SET @Parameters = N'@FullName nvarchar(60)';

EXECUTE sp_executesql @Query, @Parameters, @FullName = @FullName;
```

Using `sp_executesql` is generally considered the safest way to parameterize ad-hoc SQL queries because it does a good job of parameterizing the query and avoids 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 People.FullName, Orders.OrderDate
              FROM   Sales.Orders
                           JOIN Application.People
                                  ON Orders.ContactPersonID = People.PersonID
```


WOW! eBook
www.wowebook.org

```
              WHERE  People.FullName LIKE @FullName';
SET @Parameters = N'@FullName 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  @FullName nvarchar(60) = N'Bala Dixit';
EXECUTE sp_execute 1, @FullName;

SET @FullName = N'Bala%';
EXECUTE sp_execute 1, @FullName;
```

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, strictly from a parameterizing complex statements standpoint, is a stored procedure. Generally, the only way this makes sense as a best practice is when you have very flexible queries being executed (like in the first section of this section on ad hoc SQL, or 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 Procedures" 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 ideally 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

WOW! eBook
www.wowebook.org

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, minimally go to www.wikipedia.org and search for these terms. You should build all the code you create with these concepts in mind.)

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 in turn means that making small changes to the system is just as costly as making 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.

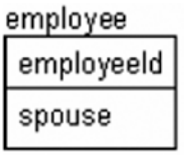


Figure 13-2. *An employee table*

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.

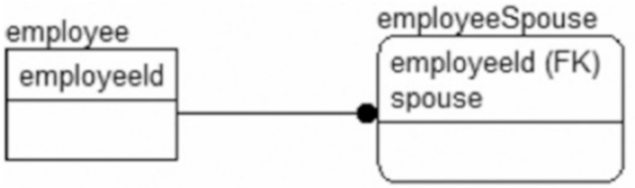


Figure 13-3. *Adding the ability to have more than one spouse*

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

WOW! eBook
www.wowebook.org

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 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 the best practice is 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 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>',16;
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);
```


WOW! eBook
www.wowebook.org

```
        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;
```

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 to set values in following statements (without some really tricky coding/forethought, as you are generating this code from other code), but starting transactions outside of the batch in which 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 usually 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. And if you are using in-memory tables, and all of your readers use SNAPSHOT isolation level, while you may never see any blocking, you could be using far more RAM than you expect due to versions being stuck in memory to satisfy your longer transactions. Bottom line: keep transactions as short as possible, in a single batch if possible.

■ **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 because of the open transaction (if the coders even think about using transactions, in which things really do get messy for the data).

WOW! eBook
www.wowebook.org

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 its own downsides:

- *Use one login for the application*: This, of course, means that you have to code your own security system for application access 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 a good way to let users use Windows Authentication, but still not have direct access to the system. The password is passed in clear text, so you will need to use encrypted connections if there are any security concerns.
- *Give the user direct access to the tables, or possibly views*: Unfortunately, this 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.

Usually, almost all applications follow the first of the three methods. Building your own login and application 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 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.

WOW! eBook
www.wowebook.org

SQL Injection

A big risk with ad hoc queries is 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 security context that executes the statements 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 because they can have some of the same issues if using dynamic SQL. 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 user-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 must 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, but that only translates to a single quote in the `EXECUTE` statement.)

```
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. The best method is to use the `QUOTENAME` function, which is there to escape names of objects and such, but works for strings as well:

```
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
```


WOW! eBook
www.wowebook.org

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 can be 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:

- By building a framework that forces you to follow the correct pattern (as do most object-relational tools, though they often force—or at least lead—you into suboptimal patterns of execution, like dealing with every statement separately without transactions.)
- By using 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 the second most important part of any computer project. It is second only to getting the right answer. While ad hoc SQL has a few benefits for performance that we have already discussed, in far more cases than not, it is far more difficult to tune when having to deal with ad hoc requests, for a few reasons:

WOW! eBook
www.wowebook.org

- *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, and tested. 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).
- *Often requires people from multiple teams*: It may seem silly, but when something is running slower, it is always SQL Server's fault (well, it is certainly blamed first in any case). With ad hoc calls, the best thing that the database administrator can do is use Extended Events to capture the queries that are executed, see if an index could help, and call for a programmer, leading to the issue in the next bullet.
- *Front-end recompile required for changing queries*: If you want to change how a query works, the app likely needs to be rebuilt and redeployed. 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 version) 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. SQL Server 2016 has a feature called the Query Store that can help you see old plans, and help to change the plan that is used. But the fact is, going in and editing a query to tune is far easier than using plan guides, particularly in light of how ad hoc SQL can change in a less manageable manner than a stored procedure.

Stored Procedures

Stored procedures are compiled batches of SQL code that can be parameterized to allow for easy reuse. The basic structure of a typical interpreted stored procedure follows. (See SQL Server Books Online at msdn.microsoft.com/en-us/library/ms187926.aspx for a complete reference.)

```
CREATE PROCEDURE <procedureName>
[(
        @parameter1  <datatype> [ = <defaultvalue> [OUTPUT]]
        @parameter2  <datatype> [ = <defaultvalue> [OUTPUT]]
        ...
        @parameterN  <datatype> [ = <defaultvalue> [OUTPUT]]
)]
AS
<T-SQL statements>
```

You can put pretty much 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. (Natively compiled procedures have far more limits, which I will cover in more detail in a later subsection of this section.) 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 because they have been deprecated, and perhaps more importantly, they are pretty horrible. (You can also not return a table-valued parameter.) After the `AS`, you can execute any T-SQL commands you need to, using

WOW! eBook
www.wowebook.org

the parameters like variables. I will largely ignore natively compiled stored procedures for now, deferring discussion to a subsection of "Advantages" that covers how using stored procedures helps you make, or prepare to make, the most of the in-memory engine.

The following is an example of a basic procedure to retrieve rows from a table (continuing to use the `WideWorldImporters` tables for these examples):

```
CREATE PROCEDURE Sales.Orders$Select
(
        @FullNameLike nvarchar(100) = '%',
        @OrderDateRangeStart date = '1900-01-01',
        @OrderDateRangeEnd date = '9999-12-31'
) AS
BEGIN
      SELECT People.FullName, Orders.OrderDate
      FROM   Sales.Orders
               JOIN Application.People
                  ON Orders.ContactPersonID = People.PersonID
      WHERE  People.FullName LIKE @FullNameLike
             --Inclusive since using Date type
        AND  OrderDate BETWEEN @OrderDateRangeStart
                           AND @OrderDateRangeEnd;
END;
```

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:

```
EXECUTE Sales.Orders$Select @FullNameLike = 'Bala Dixit';
```

Or they could use the other parameters:

```
EXECUTE Sales.Orders$Select @FullNameLike = 'Bala Dixit',
                    @OrderDateRangeStart = '2016-01-01',
                    @OrderDateRangeEnd = '2016-12-31';
```

The client doesn't know whether the database or the code is well built or even if it's horribly designed (and it is far easier to get away with bad code in a stored procedure than generated by another tier). Originally our name value might have been a part of the `Orders` table but changed to its own table when an actual data architect was hired. 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) and *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 advantages of using stored procedures as our primary interface between client and data, followed up by the pitfalls of such methods.

Advantages

The advantages of using stored procedures are considerable, and they are actually even more if you want to take full advantage of the in-memory OLTP engine. I'll discuss the following topics:

WOW! eBook
www.wowebook.org

- *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.
- *Making full use of the in-memory OLTP engine:* Requires use of natively compiled stored procedures. If your architecture is currently based on stored procedures, making this change will be far easier. I will describe a few of the limitations and differences between interpreted and natively compiled stored procedures.

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 `Sales.Orders$Select` procedure, it was 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 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 Sales.Orders$Select
...
Create temp table;
Declare cursor for (select all rows from the query 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 is horrible code to be sure. I didn't give real code so it wouldn't be confused for a positive example and imitated. However, it certainly could be built to return correct data and possibly could even be fast enough for smaller data sets ("Works on my machine!"). Even better, if we do things right, when the client executes the following code, they get the same result, regardless of the internal code:

```
EXECUTE Sales.Orders$Select @FullNameLike = 'Bala Dixit',
                            @OrderDateRangeStart = '2016-01-01',
                            @OrderDateRangeEnd = '2016-12-31';
```


WOW! eBook
www.wowebook.org

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 get metadata about the procedure we wrote earlier:

```
EXECUTE sp_describe_first_result_set
          N'Sales.Orders$Select';
```

Or even with parameters:

```
EXECUTE sp_describe_first_result_set
          N'Sales.Orders$Select @FullNameLike = ''Bala Dixit''';
```

This returns metadata about what will be returned (this is just a small amount of what is returned in the columns):

```
column_ordinal name                 system_type_name
 -------------- -------------------- ----------------------------
 1              FullName             nvarchar(50)
 2              OrderDate            date
```

For poorly formed procedures, even if it returns 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 dbo.Test (@Value int = 1)
AS
IF @value = 1
    SELECT 'FRED' as Name;
ELSE
    SELECT 200 as Name;
```

If you run the procedure with `1` for the parameter, it will return `FRED`, and 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'dbo.Test'
```

returns this (actually quite) excellent error message, letting us know that the output of this procedure can't be determined because one call may have a `varchar` type, and the next an integer:

```
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

WOW! eBook
www.wowebook.org

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 details 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 answer using the rigid requirements of precompiled stored procedures. For example, say you need a procedure that requires 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 procedure from earlier, and let's turn it into the most obvious version of a dynamic SQL statement in a stored procedure:

```
ALTER PROCEDURE Sales.Orders$Select
(
        @FullNameLike nvarchar(100) = '%',
        @OrderDateRangeStart date = '1900-01-01',
        @OrderDateRangeEnd date = '9999-12-31'
) AS
BEGIN
        DECLARE @query varchar(max) =
        CONCAT('
          SELECT People.FullName, Orders.OrderDate
          FROM   Sales.Orders
                                 JOIN Application.People
                                        ON Orders.ContactPersonID = People.PersonID
          WHERE  OrderDate BETWEEN ''', @OrderDateRangeStart, '''
                                AND ''', @OrderDateRangeEnd,'''
                                           AND People.FullName LIKE ''', @FullNameLike, '''''
);
```


WOW! eBook
www.wowebook.org

```
        SELECT @query; --for testing
        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 Sales.Orders$Select @FullNameLike = '~;''select name from sysusers--',
                                    @OrderDateRangeStart = '2016-01-01';
```

This returns three result sets: the two (including the test SELECT) from before plus a list of all of the users in the WideWorldImporters 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, you ought to use the QUOTENAME() function to insert the bracket around the name. SELECT QUOTENAME('FRED') would return [FRED].

This would look like:

```
ALTER PROCEDURE Sales.Orders$Select
(
        @FullNameLike nvarchar(100) = '%',
        @OrderDateRangeStart date = '1900-01-01',
        @OrderDateRangeEnd date = '9999-12-31'
) AS
BEGIN
        DECLARE @query varchar(max) =
        CONCAT('
      SELECT People.FullName, Orders.OrderDate
          FROM   Sales.Orders
                    JOIN Application.People
                       ON Orders.ContactPersonID = People.PersonID
          WHERE  People.FullName LIKE ', QUOTENAME(@FullNameLike,''''), '
                AND  OrderDate BETWEEN ', QUOTENAME(@OrderDateRangeStart,''''), '
                                      AND ', QUOTENAME(@OrderDateRangeEnd,''''));
         SELECT @query; --for testing
         EXECUTE (@query);
END;
```

In the next code block, I will change the procedure to safely deal with invalid quote characters, and instead of just blindly using the parameters, if the parameter value is the same as the default, I will leave off the values 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 Sales.Orders$Select
(
        @FullNameLike nvarchar(100) = '%',
```


WOW! eBook
www.wowebook.org

```
        @OrderDateRangeStart date = '1900-01-01',
        @OrderDateRangeEnd date = '9999-12-31'
) AS
BEGIN
        DECLARE @query varchar(max) =
        CONCAT('
          SELECT People.FullName, Orders.OrderDate
          FROM   Sales.Orders
                   JOIN Application.People
                      ON Orders.ContactPersonID = People.PersonID
          WHERE  1=1
          ',
           --ignore @FullNameLike parameter when it is set to all
           CASE WHEN @FullNameLike <> '%' THEN
                 CONCAT(' AND  People.FullName LIKE ', QUOTENAME(@FullNameLike,''''))
           ELSE '' END,
           --ignore @date parameters when it is set to all

           CASE WHEN @OrderDateRangeStart <> '1900-01-01' OR
                     @OrderDateRangeEnd <> '9999-12-31'
                       THEN
           CONCAT('AND  OrderDate BETWEEN ', QUOTENAME(@OrderDateRangeStart,''''), '
                                       AND ', QUOTENAME(@OrderDateRangeEnd,''''))
                       ELSE '' END);
          SELECT @query; --for testing
          EXECUTE (@query);
END;
```

Now I 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. I 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 I 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 People.FullName 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 `People.FullName` 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.)

WOW! eBook
www.wowebook.org

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 common procedures and can execute poorly formed queries that chew up important resources.

■ **Note** In general, it is best to keep most of 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 users' 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, you can do that with row-level security, but it can get kind of complex to set up block filters. If we create a stored procedure like `Contact$DeletePersonal` (meaning a contact that the user owned). Making this choice easy would be based on how well you name your procedures. I use a naming convention of `<tablename | subject area>$<action>` as you probably have witnessed throughout the book. 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 SQL" section, a lot of architects simply avoid this issue altogether (either by choice or as a result of political pressure) by letting objects connect to the database as a single user, leaving it up to the application to 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.) Note that we covered this in more detail in Chapter 9, but I want to show a quick example here as it pertains to the value of stored procedures. First, create a test user:

```
CREATE USER Fred WITHOUT LOGIN;
```

Next, create a simple stored procedure:

```
CREATE PROCEDURE dbo.TestChaining
AS
```


WOW! eBook
www.wowebook.org

```
EXECUTE ('SELECT CustomerID, StoreID, AccountNumber
          FROM   Sales.Customer');
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
The SELECT permission was denied on the object 'People', database 'WideWorldImporters',
schema 'Application'.
```

You could grant rights to the user directly to the object, but this gives users more usage than just from this procedure, which is precisely what we are trying to avoid by using stored procedures! 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::WideWorldImporters 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 = 'WideWorldImporters';`

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.YouCanDoAnything_ButDontDoThis
(
    @query nvarchar(4000)
)
WITH EXECUTE AS SELF
AS
EXECUTE (@query);
```


WOW! eBook
www.wowebook.org

This procedure gives the person who 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.YouCanDoAnything_ButDontDoThis`
- An injection-susceptible procedure, with a parameter that can hold approximately 120 characters (the length of the `dbo.YouCanDoAnything_ButDontDoThis` 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.YouCanDoAnything_ButDontDoThis` 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, can 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. As user counts go up, the amount of time begins to add up. With stored procedures, this has to be done only once.

Stored procedure parameterization uses the variables that you pass the first time the procedure is called to create the plan. This process is known as *parameter sniffing*. This is great, though sometimes you get in 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

WOW! eBook
www.wowebook.org

the plan. When you 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` hint 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 recompilation can seem a bit complicated, it really isn't; there are a few caveats, but they are generally 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 have, 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, 32GB solid-state drives, right? (Another gold star if you just said something witty about how dumb that sounded.)

The answer is obviously 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 breaking the query up into multiple queries using temporary tables as intermediate steps, 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), discuss the problem, and then rewrite and test the code (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.)

WOW! eBook
www.wowebook.org

Beyond performance, you can also fine-tune things about how the system works. For example, say you are receiving a lot of error messages, but are having a lot of trouble finding out where the error occurs. In the trigger templates, we include error logging that will capture the error that has occurred in a table. This can also be done in stored procedures. Matt Martin (www.sqletl.com) has an entire framework built around this concept of having procedures that log the error messages to a table and enabling those messages to be e-mailed when an error occurs.

The primary thing to realize is that the ability to fine-tune the data layer without affecting any changes to the other layers of code is a fantastic benefit. Of course, don't just do it without testing and proper source controls (or source control!) that you would have with any other changes. It is just quicker to regression test a single module, prove that you haven't changed the public interface (the inputs and outputs look exactly the same), and roll out to production.

Ability to Use the In-Memory Engine to Its Fullest

Of all of the arguments I have made to this point of the chapter, this one is the most compelling in my opinion, and really goes hand in hand with the previous section. As discussed in the first major section of this chapter, there are two engines in SQL Server for data storage, and there are two engines for executing SQL statements. The first is an interpreted engine that compiles your code into high-level commands that are executed and can be tweaked at runtime (hence the reason we have both an estimated plan and an actual plan we can look at when tuning queries). If you access in-memory tables using interpreted T-SQL, it accesses in interop mode. You still get the benefits of things like no lock, no blocks multivalued concurrency, but statements are still interpreted.

However, if you want to get the full potential out of the in-memory engine, you need to use natively compiled procedures. Interpreted code is interesting because it is late bound. The procedure (or function/trigger) references objects by `object_id` in the plan, but if that `object_id` changes, it can recompile by name. If you create an object and reference a name that doesn't exist, then it doesn't fail until you execute it.

Natively compiled objects are compiled into machine code, just like if you had written C# and compiled it (as are the tables you have created, by the way), and they reference all resources they use with strong connections. As such, natively compiled objects all must be created as `WITH SCHEMABINDING`, which tells you that the object cannot change when this object is referencing it. (This can make altering objects much more difficult, so it makes source control and scripts far more important.)

Every stored procedure you create natively has one other feature that is interesting. They all execute as an atomic block, which means that everything in the procedure completes, or it all fails.

The only real downside of natively compiled objects is the limitations. First, they can only access natively compiled objects and in-memory tables. Second, as of SQL Server 2016, the limitations on available syntax are quite tremendous. In the trigger examples in the downloads for Chapter 6, you will see that they are written in what would be horrible form for typical T-SQL, but the limitations on looping code are somewhat less due to the way natively compiled code of all types behaves.

For examples, I will create two quick example procedures that show you the basic format of natively compiled objects, and demonstrate the atomic block feature. This is not a programming book, so I will not dive too deep, but you should at least understand the basics so that you appreciate the discussion of the pros and cons of stored procedures, and why natively compiled objects can be tremendous for your performance—as long as you can live with the concurrency issues we discussed in Chapter 11, and the code limitations you can read about here: msdn.microsoft.com/en-us/library/dn452279.aspx. (It is quite telling that the article is, as of 2016, titled "Supported Features for Natively Compiled T-SQL Modules" rather than referring to some limitations.)

The procedures I will build will use the `Warehouse.VehicleTemperatures` table in the `WideWorldImporters` database we have been using in previous chapters. The first, very simple procedure selects rows from the `VehicleTemperatures` table where a temperature is within a range, based on a couple of parameters:

WOW! eBook
www.wowebook.org

```
USE WideWorldImporters;
GO
CREATE PROCEDURE Warehouse.VehicleTemperatures$Select
(
        @TemperatureLowRange decimal(10,2) = -99999999.99,
        @TemperatureHighRange decimal(10,2) = 99999999.99
)
WITH SCHEMABINDING, NATIVE_COMPILATION  AS
  BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'us_english')
        SELECT VehicleTemperatureID, VehicleRegistration,
               RecordedWhen, Temperature
        FROM   Warehouse.VehicleTemperatures
        WHERE  Temperature BETWEEN @TemperatureLowRange AND @TemperatureHighRange
        ORDER BY RecordedWhen DESC; --Most Recent First
  END;
```

There are a few interesting things to point out. I discussed SCHEMABINDING already, and NATIVE_COMPILATION is obviously telling the engine to use the native compilation engine. BEGIN ATOMIC is a new construct, and it basically says this entire object operates as an atomic unit. If one thing fails, the whole thing does. You specify an isolation level for the atomic block, SNAPSHOT, REPEATABLE READ, or SERIALIZABLE, and they work just like described in Chapter 11. Finally, you have to give it a language to compile under, because it doesn't default to the local language. This goes to the compiler, just as if you were building your own DLL in a less awesome language.

I compile this and then can execute the procedures just like any other stored procedure:

```
EXECUTE Warehouse.VehicleTemperatures$Select ;
EXECUTE Warehouse.VehicleTemperatures$Select @TemperatureLowRange = 4;
EXECUTE Warehouse.VehicleTemperatures$Select @TemperatureLowRange = 4.1,
                                             @TemperatureHighRange = 4.1;
```

Each of these calls will return a result set shaped like the following:

```
VehicleTemperatureID VehicleRegistration  RecordedWhen                Temperature
-------------------- -------------------- --------------------------- ---------------
65270                WWI-321-A            2016-05-30 10:04:45.0000000 4.10
65227                WWI-321-A            2016-05-30 09:18:14.0000000 4.10
64942                WWI-321-A            2016-05-29 12:37:20.0000000 4.10
64706                WWI-321-A            2016-05-29 07:44:12.0000000 4.10
...                  ...                  ...                         ...
```

Next, I will build a stored procedure that will change a row of data, with a bit of trick code to cause an error if a 0 variable is passed in, causing an error:

```
CREATE PROCEDURE Warehouse.VehicleTemperatures$FixTemperature
(
        @VehicleTemperatureID int,
        @Temperature decimal(10,2)
)
WITH SCHEMABINDING, NATIVE_COMPILATION AS
--Simulating a procedure you might write to fix a temperature that was found to be
```


WOW! eBook
www.wowebook.org

```
--outside of reasonability
  BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'us_english')
    BEGIN TRY
                --Update the temperature
                UPDATE Warehouse.VehicleTemperatures
                SET       Temperature = @Temperature
                WHERE  VehicleTemperatureID = @VehicleTemperatureID;

                --give the ability to crash the procedure for demo
                --Note, actually doing 1/0 is stopped by the compiler
                DECLARE @CauseFailure int
                SET @CauseFailure =  1/@Temperature;

                --return data if not a fail
                SELECT 'Success' AS Status, VehicleTemperatureID,
                           Temperature
                FROM   Warehouse.VehicleTemperatures
                WHERE  VehicleTemperatureID = @VehicleTemperatureID;
        END TRY
        BEGIN CATCH
                --return data for the fail
                SELECT 'Failure' AS Status, VehicleTemperatureID,
                           Temperature
                FROM   Warehouse.VehicleTemperatures
                WHERE  VehicleTemperatureID = @VehicleTemperatureID;

                THROW; --This will cause the batch to stop, and will cause this
                       --transaction to not be committed. Cannot use ROLLBACK
                           --does not necessarily end the transaction, even if it ends
                           --the batch.
        END CATCH;
  END;
```

I included the THROW in the CATCH block to show the state of the data, and to show that the TRY...CATCH construct works, and you can try things like logging errors, or whatever you want to try. To show it working, I can use the following batches:

```
--Show original value of temperature for a given row
SELECT Temperature
FROM   Warehouse.VehicleTemperatures
WHERE  VehicleTemperatureID = 65994;
```

This shows the original value:

```
Temperature
---------------------------------------
4.18
```


WOW! eBook
www.wowebook.org

Now, I execute the procedure with a value that will work:

```
EXECUTE Warehouse.VehicleTemperatures$FixTemperature
                                    @VehicleTemperatureId = 65994,
                                    @Temperature = 4.2;
```

This will return

```
Status  VehicleTemperatureID Temperature
------- -------------------- ------------------
Success 65994                4.20
```

So that worked just fine. Now, I cause an error by sending a 0 value:

```
EXECUTE Warehouse.VehicleTemperatures$FixTemperature
                                @VehicleTemperatureId = 65994,
                                @Temperature = 0;
```

This causes the following error, though you can see the data was updated successfully:

```
Status  VehicleTemperatureID Temperature
------- -------------------- ---------------
Failure 65994                0.00
Msg 8134, Level 16, State 0, Procedure VehicleTemperatures$FixTemperature, Line 18
Divide by zero error encountered.
```

Next, I use the ThrowErrorFlag to show what happens when I ignore the error:

```
EXECUTE Warehouse.VehicleTemperatures$FixTemperature
                                @VehicleTemperatureId = 65994,
                                @Temperature = 0,
                                @ThrowErrorFlag = 0;
```

It still says failure, but provides no error message:

```
Status  VehicleTemperatureID Temperature
------- -------------------- ---------------
Failure 65994                0.00
```

I check the actual value in the table, and it is 0.00. So the error from the THROW statement killed the batch and the transaction. I execute the code to set the data to 4.20, and make sure it is.

Next, let's look at what happens inside a transaction. I start a transaction, let the error occur, and see if I am still in a transaction:

```
SELECT @@TRANCOUNT AS TranStart;
BEGIN TRANSACTION
EXECUTE Warehouse.VehicleTemperatures$FixTemperature
                                @VehicleTemperatureId = 65994,
                                @Temperature = 0,
```


WOW! eBook
www.wowebook.org

```
                              @ThrowErrorFlag = 1;
GO
SELECT @@TRANCOUNT AS TranEnd;
GO
```

The data is updated, we don't rollback in the procedure, we just throw an error. The transaction that is started externally is not rolled back:

```
TranStart
-----------
0
Status  VehicleTemperatureID Temperature
------- -------------------- ---------------------------------------
Failure 65994                0.00
Msg 8134, Level 16, State 0, Procedure VehicleTemperatures$FixTemperature, Line 18
Divide by zero error encountered.
TranEnd
-----------
1
```

Now, before I have rollback, I check the value:

```
SELECT Temperature
FROM   Warehouse.VehicleTemperatures
WHERE  VehicleTemperatureID = 65994;
```

The value has not changed, because the error was thrown:

```
Temperature
---------------------------------------
4.20
```

If I make the procedure ignore the error, the data would be changed. I can then roll it back with the ROLLBACK command.

So, programming natively compiled code is very much the same as normal T-SQL, but there are some subtle, yet big differences you will need to contend with. For some things that you cannot use natively compiled code for, you can use a wrapper of interpreted code. For example, if you wanted to use a SEQUENCE object for a surrogate key, the pseudocode might be as follows:

```
CREATE PROCEDURE interpretedCode
   @columns datatype
AS --ignoring any error handling you might do
   DECLARE @surrogateKeyValue int = NEXT VALUE FOR NotNative_SEQUENCE;
   EXEC nativeCode @surrogateKeyValue = @surrogateKeyValue
                   @columns = @columns;
```

This way you can do as much natively as possible, but still use the constructs you need that are not currently supported.

WOW! eBook
www.wowebook.org

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, the high initial effort to create procedures is most often the straw that breaks the camel's proverbial back in the argument for or against stored procedures. For every time I've failed to get stored procedure access established as the method of access, this has been 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. And with the new natively compiled procedures a real possibility to migrate to in the coming years, making stored procedures a point of emphasis (even using a hybrid approach, using native code where you can, and interpreted code where you must) is a great thing to target.

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 new programmer comes in and promises fantastical 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 SQL" section, 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 '%'`, but that isn't necessarily the best-performing construct either, since a user could pass in `'%value%'` and then it would need to scan all rows, so the plan has to be ready to do "good enough." You can even use additional parameters to state `@returnAllTypeFlag` to return all rows of a certain type.

WOW! eBook
www.wowebook.org

It isn't possible to come up with a perfect 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 SQL" 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 handle the situation. Using 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 `Chapter13` database):

```
CREATE PROCEDURE Sales.Contact$Update
(
    @ContactId   int,
    @FirstName   varchar(30),
    @LastName    varchar(30),
    @CompanyName varchar(100),
    @SalesLevelId  int,
    @ContactNotes  varchar(max)
)
AS
    DECLARE @entryTrancount int = @@TRANCOUNT;

    BEGIN TRY
          UPDATE Sales.Contact
          SET    FirstName = @FirstName,
                 LastName = @LastName,
                 CompanyName = @CompanyName,
                 SalesLevelId = @SalesLevelId,
                 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;
```


WOW! eBook
www.wowebook.org

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, suppose that the SalesLevelId column is a very important column for the corporate sales process and needs to validate in the sales data if the customer should, in fact, actually be at the level assigned. 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 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 SalesLevelId 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,
    @ContactNotes  varchar(max)
)
WITH EXECUTE AS SELF
AS
    DECLARE @entryTrancount int = @@trancount;

    BEGIN TRY
       --declare variable to use to tell whether to include the sales level
       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 + ',
                    ContactNotes = ' + QUOTENAME(@ContactNotes,'''') + '

         WHERE  ContactId = ' + CAST(@ContactId AS varchar(10)) ;
         EXECUTE (@query);
    END TRY
    BEGIN CATCH
      IF @@TRANCOUNT > 0
           ROLLBACK TRANSACTION;
```


WOW! eBook
www.wowebook.org

```
      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.

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.Ccontact 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
```


WOW! eBook
www.wowebook.org

```
            SET    SalesLevelId = inserted.SalesLevelId
            FROM   Sales.Contact
                      JOIN inserted
                           ON inserted.ContactId = Contact.ContactId

           --this correlated subquery checks for values that have changed
           WHERE  EXISTS (SELECT *
                          FROM   deleted
                          WHERE  deleted.ContactId =
                                          inserted.ContactId
                            AND  deleted.SalesLevelId <>
                                          inserted.SalesLevelId)
   END TRY
   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.

Stored Procedure or Ad Hoc?

If the opinions in this section on stored procedures and ad hoc SQL 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.

WOW! eBook
www.wowebook.org

The cons of ad hoc SQL access 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.

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. And 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 you specifically ask otherwise with a `RECOMPILE` hint or object setting).
- Performance tuning is made simpler, due to the ability to know the surface area to tune. Tuning can be done without the client knowing the difference.
- Taking this performance-tuning argument one step further, stored procedures will help you make the leap to fully compiled native code in the future, which can increase throughput an order of magnitude.

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 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.

WOW! eBook
www.wowebook.org

I must also note 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.

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 a nerf-herding 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 with 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 reasonably new technology.

In most of this chapter, I have 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 the 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.

WOW! eBook
www.wowebook.org

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 that may not be possible with the new natively compiled objects due to their inability to access on-disk data. 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 the most feared new feature of SQL Server. As adoption of SQL Server versions 2005 and greater is 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/functional 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.
- *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.
- *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.

WOW! eBook
www.wowebook.org

- *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 editions 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 makes 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 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.

I am not going to provide examples of CLR coding, because it really isn't something that you will need for most implementations in a relational database. If you want to know more about the SQLCLR, I would suggest any of Adam Machanic's writing/presentations on the subject (`sqlblog.com/blogs/adam_machanic/archive/tags/SQLCLR/default.aspx`), Itzik Ben-Gan et al.'s *T-SQL Querying* (Microsoft Press, 2015), or Solomon Rutzky's Stairway to SQLCLR series on SQL Server Central (`www.sqlservercentral.com/stairway/105855/`).

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.

WOW! eBook
www.wowebook.org

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 very nerdy mob with cell phone 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.
- *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.

WOW! eBook
www.wowebook.org

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 made 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.

WOW! eBook
www.wowebook.org

CHAPTER 14

Reporting Design

Sometimes the questions are complicated and the answers are simple.

—Dr. Seuss (Theodor Seuss Geisel), American writer and cartoonist

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! Once at work, reports include enterprise dashboards, system outage reports, and timesheet cards. Any way you slice it, reporting is something that everyone uses.

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 three types of reporting you may encounter: analytical, aggregation, and operational. You will learn more about each area, including the types of modeling for each type and how to get started on your own models. Finally, you will look at common queries and structures 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 report modeling, 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 all of these purposes are valid reporting options, each option 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?" (`www.information-management.com/issues/19990901/1372-1.html`), *inny* people are most interested in getting the data into the tables in the most efficient and quick way. On the other hand, *outty* people are most interested in making the output of the data as simple as possible for the end user. Both types of people work in relational and 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.

The reporting styles that I utilize to handle these reporting situations discussed in this chapter are

- Analytical
- Aggregation
- Operational

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_14

WOW! eBook
www.wowebook.org

Each modeling paradigm has its own particular way of modeling the underlying database, storing the data, and accessing the information. We will dig further into each of these reporting styles in this chapter.

Analytical Reporting

Probably the most well-known method for storing data for analytical reporting is a dimensional model. *Analytics* focuses on two areas: understanding what has happened in the past and forecasting or trending what will happen in the future. Both of these areas and associated 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.

Data warehouses or datamarts typically use dimensional models. While other methodologies exist, the two leading methodologies, and where we will focus, for creating a dimensional model are led by two amazing technologists: Ralph Kimball and Bill Inmon. While I typically use an approach more like Kimball's, both methodologies have benefits and will 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 the model combines a number of datamarts using similar dimensions to create an enterprise data warehouse. Each datamart (and in turn, the data warehouse) is created using a star or snowflake schema, which contains one or more fact tables, linked to many dimensions.

To create the bus architecture, a developer creates subject-oriented datamarts that contain business logic for a particular department. Over time, new datamarts develop, 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 `www.kimballgroup.com`. Though they have closed their doors as a unit, their website and books still have great value.

Bill Inmon

The other leading data warehouse methodology is proposed 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 datamarts 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 `www.inmoncif.com` for additional information on Bill Inmon's approach.

Aggregation Reporting

A hybrid of analytical and operational 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.

WOW! eBook
www.wowebook.org

Modeling summary tables typically occurs when a report that used an aggregated value becomes too slow for the end users. The slowdown could occur because the amount of aggregated data has increased or the end users have new unplanned questions to answer. To fix the slow reporting situation, you can create summary tables 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 whose resources can be consumed by queries and reports.

Operational Reporting

The premise of both analytical and aggregation reporting is to use a separate database from the operational application's database. In this way, reporting will not affect the operational system's performance by blocking queries or using too many resources. The opposing side to that benefit is the cost of developing the reporting database as well the time required to move the data from the operational system to the reporting system, which will cause the reported data to be a bit behind that of the operational system. But what if there was a way to not affect the operational system database performance *and* still be able to access the data in real time? This chapter will show you how, using in-memory operational reporting in SQL Server 2016.

Operational reporting allows you to write queries directly on the operational system. These reporting queries run at the same time as the application system's queries and use the same database as the operational application. This method may be appealing to you if you need access to the real-time data from the operational system or if you have limited bandwidth to support a separate reporting database. If this method sounds appealing, you will be interested in reading this chapter's review of the additional benefits of the operational reporting system and how best to set up a SQL Server database and model to utilize operational reporting.

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. Additionally, 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 folks, I am golden!

A recap of the steps listed in this section is shown in Figure 14-1.

WOW! eBook
www.wowebook.org

Figure 14-1. *Requirements-gathering steps diagram*

No matter which reporting solution fits your scenario the best, you will still follow a similar requirements-gathering process. The phase in which you use the business process requirements to decide which type of reporting to use and how to create that particular type of data model (listed as step 4) will be described in the following sections of this chapter.

The requirements-gathering process 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 for a business user to be so focused on their department and area that they ignore a business process that doesn't affect them directly. Your goal as a reporting modeler is to learn about all areas and processes within the organization. Often, you'll discover ties between the siloed departments that were initially impossible to see!

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 table types exist, but these are the 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 dimension tables.

WOW! eBook
www.wowebook.org

We will start by describing the different types of dimensions that we can create and wrap up with the types of facts we can create. After we complete our discussion of facts and dimensions, we will end up with a complete healthcare insurer data model, as shown in Figure 14-2.

Figure 14-2. *Dimensional model for healthcare payer data*

WOW! eBook
www.wowebook.org

Dimensions

As previously stated, a dimension is the table that describes an entity. To create our dimensions, we need to start by understanding the business process associated with that entity. After we describe the business process, the next 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 the business process.

As an example, let's start with a health insurance company that pays its customers based on their claims. 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 be either 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% of camouflage pants come in green or gray, but 1% come in purple, you must know about that 1% from a data perspective.

Once we'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 we 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 be either *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 questions must be asked of and answered by the business.

We must also know from where the data will be sourced to ensure that the model uses the correct names and datatypes. Fortunately, the healthcare field 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 we have gone through the requirements process, discovered all of the business processes, and know everything there is to know about our business, we can begin 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 × 1,440 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.

WOW! eBook
www.wowebook.org

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 `20160928`, for September 28, 2016 for the United States folks. Smart keys make sense for date and time dimensions, because you know the entity itself will never change.
- *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 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 reporting 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. An example of a stored procedure that does this is provided in the next listing. Note the initial `INSERT` statement that adds the unknown row if it doesn't exist.

WOW! eBook
www.wowebook.org

In contrast to OLTP tables, *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 Procedure
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)
        BEGIN
                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
END;
```

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 '01/01/2016', '12/31/2017';
```


WOW! eBook
www.wowebook.org

A sample of the data in the table is shown here:

```
DateKey     DateValue  DayValue    WeekValue   MonthValue  YearValue
----------- ---------- ----------- ----------- ----------- -----------
-1          1900-01-01 -1          -1          -1          -1
20161230    2016-12-30 30          53          12          2016
20161231    2016-12-31 31          53          12          2016
20170101    2017-01-01 1           1           1           2017
20170102    2017-01-02 2           1           1           2017
```

This dimension and stored procedure may be exactly what you need, and if so, move on to 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 never changes. 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 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 reiterated many times, 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.

WOW! eBook
www.wowebook.org

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 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
));
```

For demonstration's sake, load the member dimension using 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])
SELECT -1, 'UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UN','UNKNOWN'
UNION ALL
SELECT 1, 'IN438973','Brandon','Jones','Dr. Keiser & Associates','Henrico','VA','<1 year'
UNION ALL
SELECT 2, 'IN958394','Nessa','Gomez','Healthy Lifestyles','Henrico','VA','1-2 year'
UNION ALL
SELECT 3, 'IN3867910','Catherine','Patten','Dr. Jenny Stevens','Spotsylvania','VA','<1
year';
GO
SET IDENTITY_INSERT [dim].[Member] OFF;
```

Use the following query to retrieve the sample data that follows:

```
SELECT MemberKey, InsuranceNumber, FirstName, LastName, PrimaryCarePhysician
FROM dim.Member;
```


WOW! eBook
www.wowebook.org

Here are the results:

```
MemberKey   Insurance   FirstName   LastName   PrimaryCarePhysician
----------- ----------- ----------- ---------- --------------------
-1          UNKNOWN     UNKNOWN     UNKNOWN    UNKNOWN
1           IN438973    Brandon      Jones      Dr. Keiser & Associates
2           IN958394    Nessa        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 where we don't care about the previous value, 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 add a flag called `isCurrent` to indicate which row contains the most current information. The code to add this column is listed here:

```
ALTER TABLE dim.Member
ADD isCurrent INTEGER NOT NULL DEFAULT 1;
```

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 Brandon changing his primary care physician and updating the old record:

```
INSERT INTO [dim].[Member]
([InsuranceNumber],[FirstName],[LastName],[PrimaryCarePhysician]
        ,[County],[StateCode],[MembershipLength])
VALUES
('IN438973','Brandon','Jones','Dr. Jenny Stevens','Henrico','VA','<1 year');
GO

UPDATE [dim].[Member] SET isCurrent = 0
WHERE InsuranceNumber = 'IN438973' AND PrimaryCarePhysician = 'Dr. Keiser & Associates';
```

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:

```
MemberKey   Insurance   FirstName   LastName   PrimaryCarePhysician          isCurrent
----------- ----------- ----------- ---------- ----------------------------- ---------
-1          UNKNOWN     UNKNOWN      UNKNOWN    UNKNOWN                       1
1           IN438973    Brandon      Jones      Dr. Keiser & Associates       0
2           IN958394    Nessa        Gomez      Healthy Lifestyles            1
3           IN3867910   Catherine    Patten     Dr. Jenny Stevens             1
4           IN438973    Brandon      Jones      Dr. Jenny Stevens             1
```


WOW! eBook
www.wowebook.org

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 more 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;
```

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;
```

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://download.cms.gov/nppes/NPI_Files.html. 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,
        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
));
```

Add an initial set of data using the following query:

```
-- Insert sample data into Provider dimension table
SET IDENTITY_INSERT [dim].[Provider] ON;
```


WOW! eBook
www.wowebook.org

```
GO
INSERT INTO [dim].[Provider]
([ProviderKey],[NPI],[EntityTypeCode],[EntityTypeDesc],[OrganizationName],
                [DoingBusinessAsName],[Street],[City],[State],[Zip],[Phone])
SELECT -1, 'UNKNOWN',-1,'UNKNOWN','UNKNOWN','UNKNOWN',
        'UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN'
UNION ALL
SELECT 1, '1234567',1,'Individual','Patrick Lyons','Patrick Lyons',
        '80 Park St.','Boston','Massachusetts','55555','555-123-1234'
UNION ALL
SELECT 2, '2345678',1,'Individual','Lianna White, LLC','Dr. White & Associates',
        '74 West Pine Ave.','Waltham','Massachusetts','55542','555-123-0012'
UNION ALL
SELECT 3, '76543210',2,'Organization','Doctors Conglomerate, Inc','Family Doctors',
        '25 Main Street Suite 108','Boston','Massachusetts','55555','555-321-4321'
UNION ALL
SELECT 4, '3456789',1,'Individual','Dr. Drew Adams','Dr. Drew Adams',
        '1207 Corporate Center','Peabody','Massachusetts','55554','555-234-1234';
SET IDENTITY_INSERT [dim].[Provider] OFF;
```

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, more normalized, 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 and each contains a set of benefits. Sometimes, we look at the metrics at the benefit level, and other times, we look at the metrics 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 dimension table creation script is listed here:

```
-- Create the Benefit dimension table
CREATE TABLE dim.Benefit(
        BenefitKey INTEGER IDENTITY(1,1) NOT NULL,
```


WOW! eBook
www.wowebook.org

```
        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
));
```

The health plan dimension table 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
CONSTRAINT PK_HealthPlan PRIMARY KEY CLUSTERED
(
        HealthPlanKey ASC
));
```

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 health 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);
```

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
SET IDENTITY_INSERT [dim].[HealthPlan] ON;
GO
INSERT INTO [dim].[HealthPlan]
  ([HealthPlanKey], [HealthPlanIdentifier],[HealthPlanName])
SELECT 1, 'BRON','Bronze Plan'
UNION ALL
SELECT 2, 'SILV','Silver Plan'
UNION ALL
SELECT 3, 'GOLD','Gold Plan';
GO
SET IDENTITY_INSERT [dim].[HealthPlan] OFF;
```


WOW! eBook
www.wowebook.org

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(s) 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*, which is simple to identify from the dimension's 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.

In our healthcare 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
));
```

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)
SELECT -1, 'UNKNWN', 'UNKNOWN'
UNION ALL
SELECT 1, 'AUTO', 'ACCEPTED'
UNION ALL
SELECT 2, 'MANUAL', 'ACCEPTED'
UNION ALL
SELECT 3, 'DENIED', 'DENIED';
GO
SET IDENTITY_INSERT dim.AdjudicationType OFF;
```


WOW! eBook
www.wowebook.org

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 the type 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 Procedure Coding System (HCPCS) codes, respectively. We can access this set of information through the Centers for Medicare and Medicaid Services web site (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(350) 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
));
```

Depending on how in-depth your business needs are, you can add more fields to the diagnosis and procedure dimensions. To download a full list of diagnoses, go to www.cms.gov/ICD10/downloads/ICD10OrderFiles.zip, and for all procedures, go to www.cms.gov/Medicare/Coding/HCPCSReleaseCodeSets/Downloads/2016-Alpha-Numeric-HCPCS-File.zip. You can also use the files from CMS to load the set of static rows. Sample SQL Server Integration Services packages are provided with the code for this chapter to load these values.

WOW! eBook
www.wowebook.org

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: for 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 the combination of 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.

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 HCPCS. 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, 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 (which means to judge whether to pay). The claim can be either 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 their manipulations 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,
```


WOW! eBook
www.wowebook.org

```
        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
);
```

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 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 or manually adjudicated amounts. We will model this

WOW! eBook
www.wowebook.org

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;
```

Listing 14-4. Rejected Query to Pull Amount Information

```
SELECT CASE dat.AdjudicationType
                WHEN 'AUTO'
                THEN fp.ClaimAmount
                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;
```

The next metric we want to discuss in more depth is the count of adjudications, which helps 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
```


WOW! eBook
www.wowebook.org

```
        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),
        0,
        0,
        0;
SET @i = @i + 1;
END;
GO
UPDATE fact.ClaimPayment
SET AutoAdjudicatedCount = CASE AdjudicationTypeKey        WHEN 1 THEN 1 ELSE 0 END
        ,ManualAdjudicatedCount = CASE AdjudicationTypeKey WHEN 2 THEN 1 ELSE 0 END
        ,DeniedCount = CASE AdjudicationTypeKey            WHEN 3 THEN 1 ELSE 0 END
FROM fact.ClaimPayment;
GO
```

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, known as *degenerate dimensions*, 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. You typically use these columns 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.

WOW! eBook
www.wowebook.org

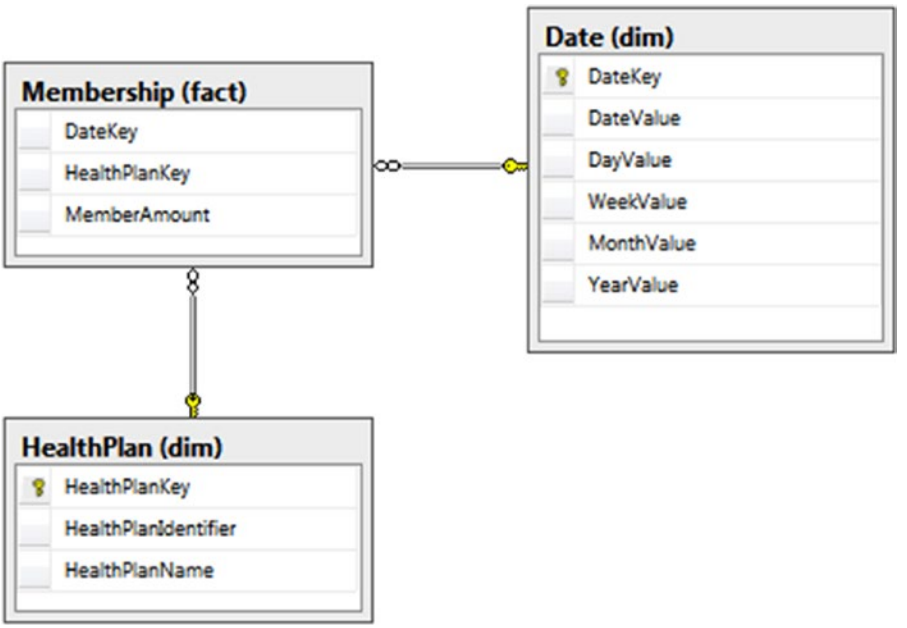


Figure 14-3. *Snapshot fact dimensional model for our healthcare payer*

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 here 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:

```
-- Create Membership snapshot fact table
CREATE TABLE fact.Membership (
        DateKey INTEGER NOT NULL,
        HealthPlanKey INTEGER NOT NULL,
        MemberAmount INTEGER NOT NULL
);
```


WOW! eBook
www.wowebook.org

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);
```

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 = '1/1/2016';
SET @enddate = '12/31/2016';

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;
```

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, imagine 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.

```
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 ('2016-09-01', '2016-09-02', '2016-09-03')
AND fm.HealthPlanKey = 2;
```


WOW! eBook
www.wowebook.org

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 the 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 = 7;
```

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 = 7;
```

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;
```


WOW! eBook
www.wowebook.org

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 '01/01/2016' and '12/31/2016'
GROUP BY dp.OrganizationName, dd.MonthValue;
```

A reporting 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;
```

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.

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 ever 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

Developers typically use indexes to increase 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:

WOW! eBook
www.wowebook.org

```
SELECT ProcedureKey, SUM(ClaimAmount) AS ClaimByProcedure
FROM fact.ClaimPayment
GROUP BY ProcedureKey;
```

With no index, this results in the execution plan in Figure 14-4.

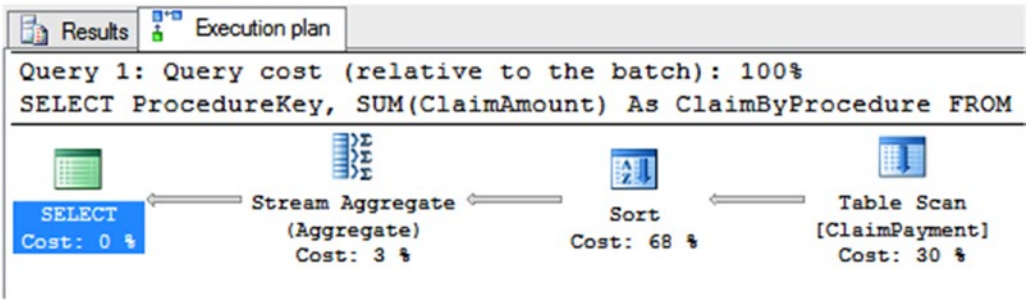


Figure 14-4. *"Before" execution plan for claim amount query on dimensional model*

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 feature provided in SQL Server called *columnstore indexes*, that were introduced in Chapter 10. Rather than store index data per row, a columnstore index stores data by column. They are typical indexes in that you can have a clustered columnstore index, which orders the data in columns, and a nonclustered columnstore index, which uses pointers to the data stored in columns, without changing the structure of the data. For each of the columns referenced in a columnstore index, all of the data is grouped and stored together. Starting in SQL Server 2016, clustered and nonclustered columnstore indexes are updateable, making them easier than ever to maintain. Create a columnstore index using the following script:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX ColumnStoreIndex ON fact.ClaimPayment
(
        DateKey,
        MemberKey,
        AdjudicationTypeKey,
        ProviderKey,
        DiagnosisKey,
        ProcedureKey,
        ClaimID,
        ClaimAmount,
        AutoPayoutAmount,
        ManualPayoutAmount,
```


WOW! eBook
www.wowebook.org

```
        AutoAdjudicatedCount,
        ManualAdjudicatedCount,
        DeniedCount
);
```

When we run the query again, we can see that the plan has changed quite drastically, as shown in Figure 14-5.

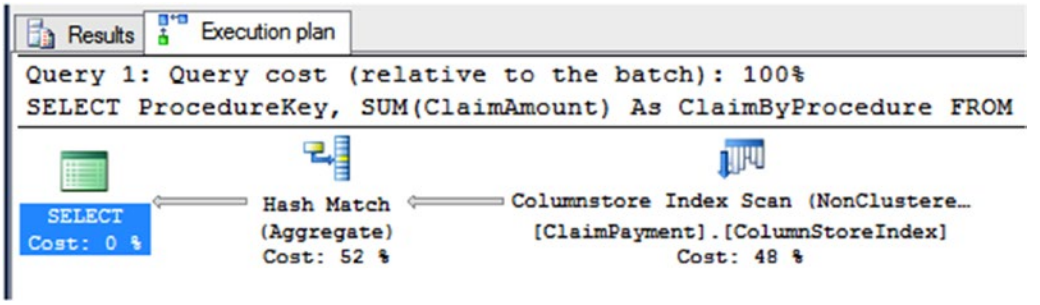


Figure 14-5. *"After" execution plan for claim amount query on dimensional model*

The columnstore index scan operator (rather than the table scan operator from the original query) shows that this query is working with a columnstore index.

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. 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:

- Speed is increased due to the preaggregated values.
- Fewer joins equates to ease of maintenance and understandability.

Before jumping into this solution, you'll want to consider some disadvantages as well:

- A 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.

WOW! eBook
www.wowebook.org

Initial Summary Table

Once you've decided that the benefits of summary modeling 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 be either 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 14-6.

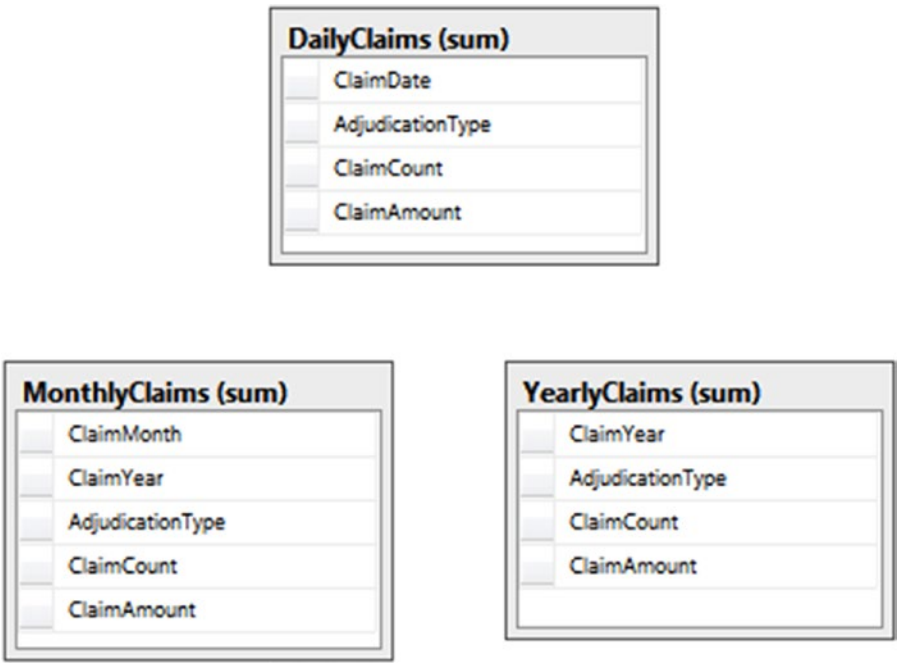


Figure 14-6. *Summary model for healthcare claim data*

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
```


WOW! eBook
www.wowebook.org

```
-- 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 we load the data, we need to preaggregate the values to match our grain. For example, our claims system probably records more than one claim per day, but we want to see all claims together! We can use SQL statements with SUM functions or SQL Server Integration Services packages with the aggregate task, but either way, our 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
---------- ---------------- ----------- ---------------------------------------
2016-04-23 AUTO             1           54103.93
2016-05-05 DENIED           1           30192.30
2016-04-26 MANUAL           1           9344.87
2016-07-06 DENIED           1           29994.54
2016-05-25 AUTO             1           52412.47
```


WOW! eBook
www.wowebook.org

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:

```
-- Insert 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.

WOW! eBook
www.wowebook.org

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.

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 timeframe the users are interested in, they will use one of the following very simple queries:

```
SELECT ClaimDate, AdjudicationType, ClaimCount, ClaimAmount
FROM sum.DailyClaims;

SELECT ClaimMonth, ClaimYear, AdjudicationType, ClaimCount, ClaimAmount
FROM sum.MonthlyClaims;

SELECT ClaimYear, AdjudicationType, ClaimCount, ClaimAmount
FROM sum.YearlyClaims;
```

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(MONTH, -3, GETDATE()) and GETDATE()
ORDER BY ClaimDate;
```

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;
```

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 14-7.

WOW! eBook
www.wowebook.org

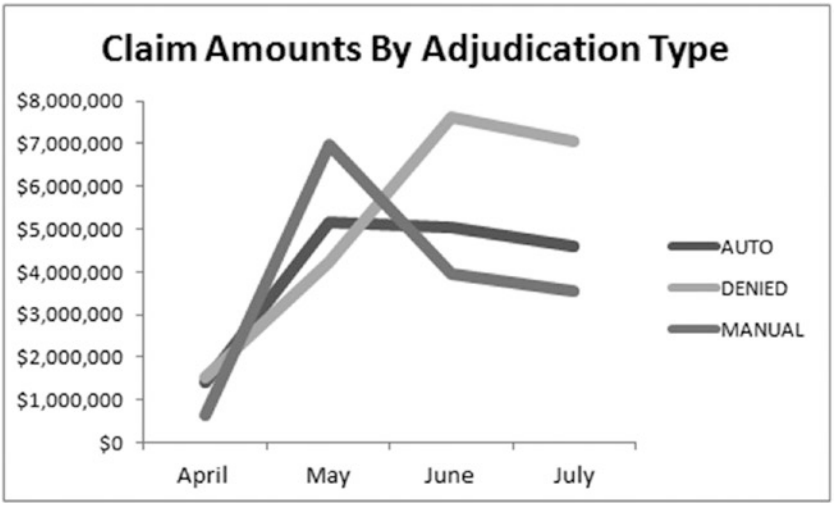


Figure 14-7. *Sample claim amount report*

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 = '07/10/2016';
```

With no index, we get a fairly poor plan with a table scan and stream aggregate, as shown in Figure 14-8.

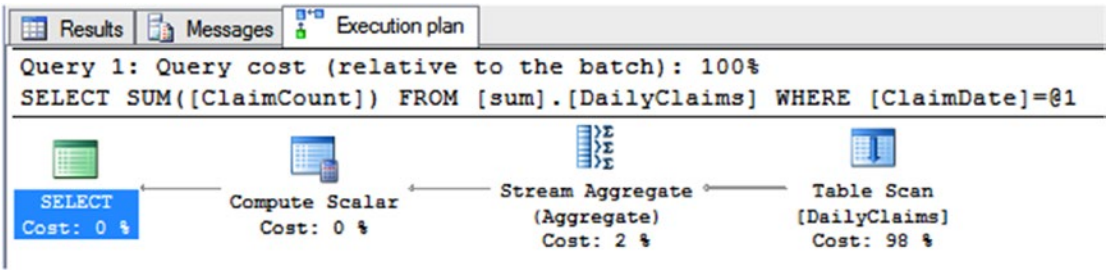


Figure 14-8. *"Before" execution plan for claim count query on summary model*

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,
```


WOW! eBook
www.wowebook.org

```
        ClaimCount
);
GO
```

We end up with the plan in Figure 14-9, in which the index seek definitely improves our performance.

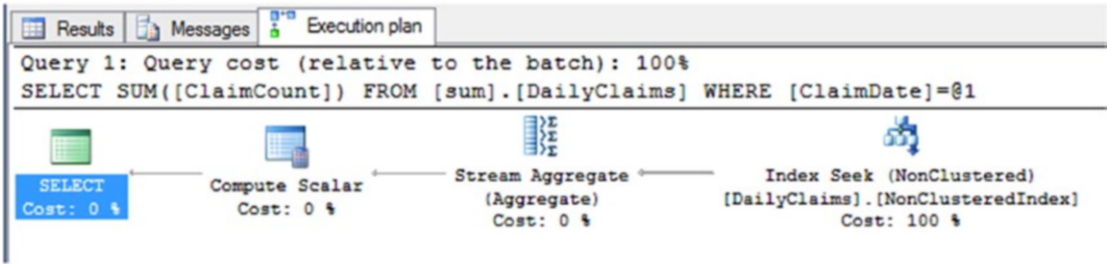


Figure 14-9. *"After" execution plan for claim count query on summary model*

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!

Modeling (or Lack Thereof) for Operational Reporting

The third option to cover is operational reporting. As previously described, one of the benefits of operational reporting is that you *don't* have to model anything. You can use your existing, normalized, transactional model to write queries. We can do this by using columnstore indexes applied to our relational tables. Analytic queries use the columnstore index, while OLTP queries use the rowstore indexes. This section will explain the benefits and disadvantages of using operational reporting and describe how to set up your database to implement operational reporting. We will use memory optimized tables for the example since they provide the best concurrency, but this is available for on-disk structures as well.

Similar to the previous reporting models we discussed, you should still walk through the requirements-gathering process as described earlier in the chapter (refer to Figure 14-1 for a refresher). This ensures that you have captured all of the needed reporting requirements and are not overlooking a scenario where the reporting would be better captured by another of the reporting models. Once you've reached the fourth step of the process, you have enough information to decide what type of reporting you want to use.

Although in the past we have typically recommended that reporting be done on a separate database than the operational system, in SQL Server 2016, we now have the option of writing queries directly against the operational database. This provides many benefits, including:

- The ability to consume and react to information from the database more quickly than other methods
- Minimal development and maintenance overhead, as compared to having to write extract, transform, and load (ETL) packages for the other methods
- No additional storage requirements that an additional database would need

I do have to caution you that operational reporting will not work in every scenario. For example, if you need to combine data sources or use data sources that are not SQL Server, this will not work. Additionally, if you need to pre-aggregate your data in many different ways, using an analytical dimensional model or aggregation model may serve your needs the best. If you've reached step 4 of the process and still believe operational reporting is the way to go, read on!

WOW! eBook
www.wowebook.org

Sample Operational Data Model

To fully understand the benefits of the operation reporting model, let's start by examining a simple normalized data model that is used to store the claims and adjudication information. As shown in Figure 14-10, we have three different tables that were created for the transactional system to record the data as it is entered by a system and the customer service representatives:

- `AdjudicationType`: This table contains a field for the surrogate key, which has no meaning to the business but will help for easy joins with other tables within the model. The second column is the value of the type (either Automatic, Manual, or Declined).
- `Claim`: This table contains information about the claim, including several dates, a link to the adjudication type, and a link to the `Member` table.
- `Member`: This table contains information about the member, including a surrogate key, business identifier, and related information about the member.

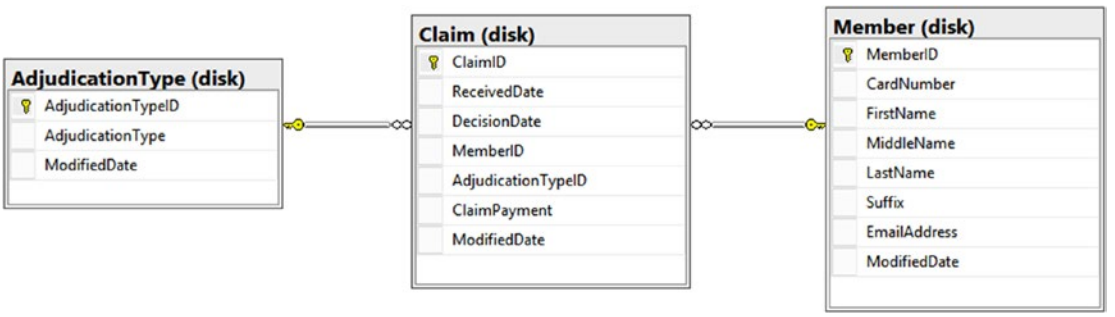


Figure 14-10. *Normalized model for healthcare claims data*

In-Memory OLTP

To use the in-memory OLTP feature for reporting, you'll need to set up the tables as memory-optimized tables in SQL Server. Microsoft originally introduced in-memory OLTP in SQL Server 2014, but has greatly enhanced the feature in SQL Server 2016. Memory-optimized tables are meant to increase the speed of transactional processing, while ensuring durability of transactions. Using the power of in-memory OLTP can greatly help your reporting needs.

Before putting your tables into memory, you must properly set up your database to allow it to manage its in-memory objects. Perform the following steps:

1. Ensure your compatibility mode is set to SQL Server 2016.
2. Turn on the `MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT` configuration setting in order to make DDL operations not need to specify SNAPSHOT isolation level in explicit transactions.
3. Create a filegroup that specifies that it can contain memory-optimized data.

For more information on these steps and specific syntax, see `https://msdn.microsoft.com/en-us/library/dn133186.aspx?f=255&MSPPError=-2147217396`. Once the database is prepared, you can create your memory-optimized tables.

WOW! eBook
www.wowebook.org

To create the tables in our healthcare claims data model, run the following script:

```
--Create In-memory OLTP tables
CREATE TABLE dbo.AdjudicationType(
        AdjudicationTypeID int IDENTITY(1,1) NOT NULL PRIMARY KEY NONCLUSTERED,
        AdjudicationType varchar(50) NOT NULL,
        ModifiedDate datetime NOT NULL
) WITH
        (MEMORY_OPTIMIZED = ON,
        DURABILITY = SCHEMA_AND_DATA);
GO

CREATE TABLE dbo.Member(
        MemberID int IDENTITY(1,1) NOT NULL PRIMARY KEY NONCLUSTERED,
        CardNumber varchar(10) NOT NULL,
        FirstName varchar(50) NOT NULL,
        MiddleName varchar(50) NULL,
        LastName varchar(50) NOT NULL,
        Suffix varchar(10) NULL,
        EmailAddress varchar(40) NULL,
        ModifiedDate datetime NOT NULL
) WITH
        (MEMORY_OPTIMIZED = ON,
        DURABILITY = SCHEMA_AND_DATA);
GO

CREATE TABLE dbo.Claim(
        ClaimID int IDENTITY(1,1) NOT NULL PRIMARY KEY NONCLUSTERED,
        ReceivedDate datetime NOT NULL,
        DecisionDate datetime NOT NULL,
        MemberID int NULL,
        AdjudicationTypeID int NOT NULL,
        ClaimPayment money NOT NULL,
        ModifiedDate datetime NOT NULL
) WITH
        (MEMORY_OPTIMIZED = ON,
        DURABILITY = SCHEMA_AND_DATA);
GO

ALTER TABLE dbo.Claim  WITH CHECK
ADD  CONSTRAINT FK_dboClaim_AdjudicationType FOREIGN KEY(AdjudicationTypeID)
REFERENCES dbo.AdjudicationType (AdjudicationTypeID);
GO

ALTER TABLE dbo.Claim  WITH CHECK
ADD  CONSTRAINT FK_dboClaim_Member FOREIGN KEY(MemberID)
REFERENCES dbo.Member (MemberID);
GO
```


WOW! eBook
www.wowebook.org

Now that we have created the objects, we can use the following scripts to load the data:

```
--Load AdjudicationType Table
INSERT INTO AdjudicationType VALUES ('AUTO', GETDATE());
GO

INSERT INTO AdjudicationType VALUES ('MANUAL', GETDATE());
GO

INSERT INTO AdjudicationType VALUES ('DENIED', GETDATE());
GO

--Load Member Table
INSERT INTO [dbo].[Member] ([CardNumber], [FirstName], [MiddleName],
               [LastName], [Suffix], [EmailAddress], [ModifiedDate])
VALUES ('ANT48963', 'Jessica', 'Diane', 'Moss', 'Ms.', 'jessica@email.com', GETDATE());
GO

INSERT INTO [dbo].[Member] ([CardNumber], [FirstName], [MiddleName],
               [LastName], [Suffix], [EmailAddress], [ModifiedDate])
VALUES ('ANT8723', 'Richard', 'John', 'Smith', 'Mr.', 'richard@email.com', GETDATE());
GO

INSERT INTO [dbo].[Member] ([CardNumber], [FirstName], [MiddleName],
               [LastName], [Suffix], [EmailAddress], [ModifiedDate])
VALUES ('BCBS8723', 'Paulette', 'Lara', 'Jones', 'Mrs.', 'paulette@email.com', GETDATE());
GO

--Load Claim Table
DECLARE @i AS INT;
SET @i = 0;

WHILE @i < 250000
BEGIN
INSERT INTO [dbo].[Claim] ([ReceivedDate], [DecisionDate], [MemberID]
           ,[AdjudicationTypeID], [ClaimPayment], [ModifiedDate])
VALUES (DATEADD(day, CAST((RAND()*100) AS int) % 28 + 1
        , DATEADD(month, CAST((RAND()*100) AS int) % 12 + 1, '2016-01-01'))
        , DATEADD(day, CAST((RAND()*100) AS int) % 28 + 1
        , DATEADD(month, CAST((RAND()*100) AS int) % 12 + 1, '2016-01-01'))
        , CAST((RAND()*100) AS int) % 3 + 1
        , CAST((RAND()*100) AS int) % 3 + 1
        , CAST((RAND()*1000) AS decimal(5, 2))
        , GETDATE());
SET @i = @i + 1;
END;
GO
```


WOW! eBook
www.wowebook.org

If you are using any stored procedures to only access the memory-optimized tables, you ought to set those as natively compiled stored procedures , but interpreted objects can access memory optimized tables with only the changes discussed in Chapter 13. Once everything is set up, the transactional queries will run faster, and you can begin querying!

Operational Querying

The exciting part about setting up the operational database tables as memory optimized is running our reporting queries on that same database. The ability to query the operational database tables gives us immediate feedback on our data. Let's look at the kind of queries we may want to run on the operational database and then look at how we can optimize them using in-memory columnstore indexes.

Queries

Let's use queries that we discussed earlier in this chapter to show the power of using in-memory OLTP. The first business question that we need to answer (which we also looked at in the "Analytical Querying" section) is, "How many claims have had a denied payment?" We can answer that questions using the following query:

```
SELECT COUNT(m.CardNumber) AS ClaimCount
FROM dbo.Claim AS c
LEFT JOIN dbo.AdjudicationType AS adj ON c.AdjudicationTypeID = adj.AdjudicationTypeID
LEFT JOIN dbo.Member AS m ON c.MemberID = m.MemberID
WHERE AdjudicationType = 'DENIED';
```

The second business question we need to answer is, "How much money in claims was approved automatically, approved manually, or denied over the past year?" This question is a variation of what was covered in the summary reporting section, and the answer can be written as a query against the in-memory OLTP system. The query we can use is

```
SELECT AdjudicationType, SUM(ClaimPayment) AS TotalAmt
FROM dbo.Claim AS c
LEFT JOIN dbo.AdjudicationType AS adj ON c.AdjudicationTypeID = adj.AdjudicationTypeID
LEFT JOIN dbo.Member AS m on c.MemberID = m.MemberID
WHERE DecisionDate > DATEADD(year, -1, getdate())
GROUP BY AdjudicationType;
```

These queries are fairly straightforward and can be written easily by any developer who is familiar with the original OLTP database. Next, we'll look at how to use indexing to improve the performance of these queries.

Indexing

Once the structures are in place and populated with data, we will look at one of the indexing options recommended for memory-optimized tables and operational reporting. The operational database tables most likely will already have a series of indexes, both clustered and nonclustered, that assist in the performance of queries used for the operational database. To handle our operational reporting queries, we will add more indexes.

In this scenario, we will use a clustered columnstore index (for on-disk tables, you use non-clustered). The columnstore index uses a columnar format to shrink the values within a column and provide an indexing structure to allow the SQL Server engine to access the appropriate row as quickly as possible. Combining a

WOW! eBook
www.wowebook.org

columnstore index with the in-memory capability that we just discussed makes a very powerful combination! For the operational healthcare data model described here, we will get the most benefit out of adding a columnstore index to the claim table.

Before that, take a look at the execution plans for our two queries on the same table and data, but on disk instead of memory-optimized. The execution plan for the query "How many claims have had a denied payment?" is shown in Figure 14-11, and the execution plan for the query "How much money in claims was approved automatically, approved manually, or denied over the past year?" is shown in Figure 14-12.

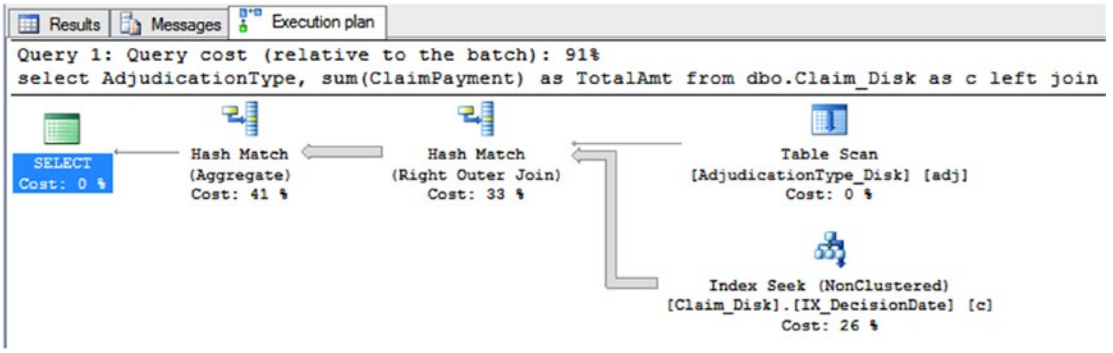


Figure 14-11. *"Before" execution plan for denied claim count query on operational model*

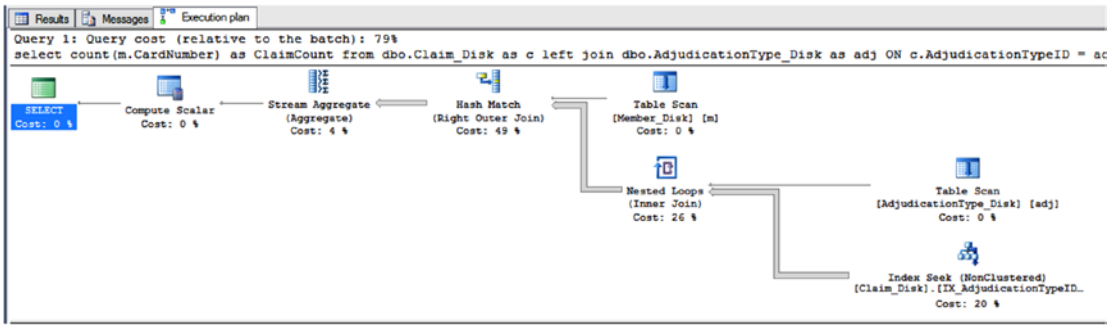


Figure 14-12. *"Before" execution plan for claim amount by type query on operational model*

Now we are ready to create the columnstore index on our main table, `dbo.Claim`. Run the following `CREATE` script to create a memory-optimized table with a columnstore index:

```
--Create Columnstore Tables
CREATE TABLE dbo.Claim_Columnstore
(
        ClaimID int IDENTITY(1,1) NOT NULL,
        ReceivedDate datetime NOT NULL,
        DecisionDate datetime NOT NULL,
        MemberID int NULL,
        AdjudicationTypeID int NOT NULL,
        ClaimPayment money NOT NULL,
        ModifiedDate datetime NOT NULL,
```


WOW! eBook
www.wowebook.org

```
 PRIMARY KEY NONCLUSTERED
(
        ClaimID ASC
),
INDEX IX_COLUMNSTORE CLUSTERED COLUMNSTORE
)WITH ( MEMORY_OPTIMIZED = ON , DURABILITY = SCHEMA_AND_DATA );
GO

ALTER TABLE dbo.Claim_Columnstore  WITH CHECK
ADD  CONSTRAINT FK_dboClaimColumnstore_AdjudicationType
FOREIGN KEY(AdjudicationTypeID)
REFERENCES dbo.AdjudicationType (AdjudicationTypeID);
GO

ALTER TABLE dbo.Claim_Columnstore
CHECK CONSTRAINT FK_dboClaimColumnstore_AdjudicationType;
GO

ALTER TABLE dbo.Claim_Columnstore  WITH CHECK
ADD  CONSTRAINT FK_dboClaimColumnstore_Member
FOREIGN KEY(MemberID)
REFERENCES dbo.Member (MemberID);
GO

ALTER TABLE dbo.Claim_Columnstore
CHECK CONSTRAINT FK_dboClaimColumnstore_Member;
GO
```

Looking at the execution plan for both of our queries, as shown in Figures 14-13 and 14-14, you can see that the query uses our new index, returning the information quickly and without disturbing any pre-existing operational queries.

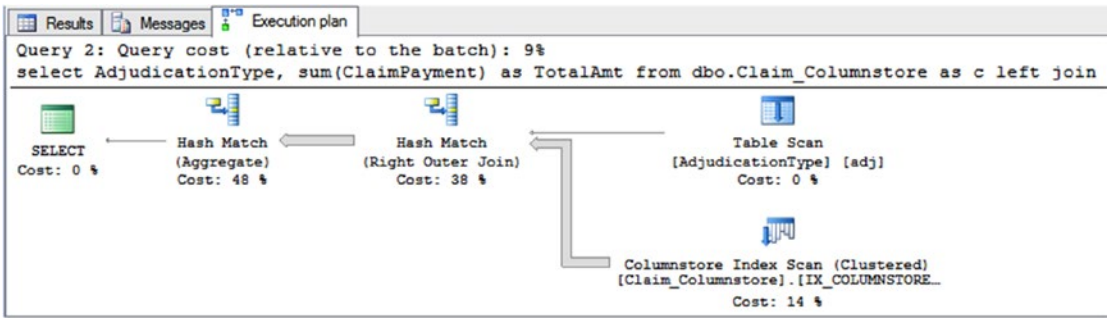


Figure 14-13. *"After" execution plan for denied claim count query on operational model*

WOW! eBook
www.wowebook.org

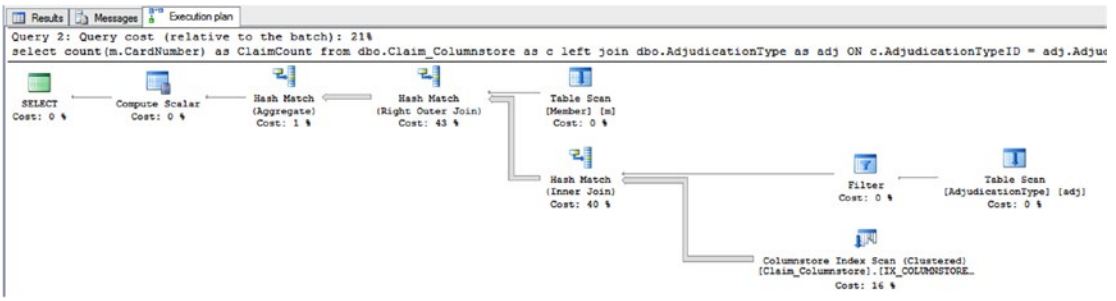


Figure 14-14. *"After" execution plan for claim amount by type query on operational model*

Some queries have seen over ten times performance improvement by utilizing in-memory OLTP and columnstore indexes!

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 could model a reporting database differently than a transactional system and the different ways you can model for reporting. I definitely did not cover every potential scenario you may face, instead, this was an introduction to get started with some initial reporting structures.

Dimensional modeling for analytical reporting, summary modeling for aggregate reporting, and operational 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, operational reporting can be a great option if minimizing development time and reducing resources are important to you. We covered when and how you could use the operational reporting option as compared to dimensional and analytical modeling.

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's questions, you have no need for reporting!

WOW! eBook
www.wowebook.org

APPENDIX A

Scalar Datatype Reference

Choosing proper datatypes to match the domain to satisfy 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 from $-10^{38} + 1$ 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.

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7_15

WOW! eBook
www.wowebook.org

    - `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 2GB.
    - `text`: Large text values; maximum length of $2^{31} - 1$ (2,147,483,647) bytes, or 2GB. (Note that this datatype is outdated, *officially* deprecated, 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 2GB.
    - `image`: Large binary data; maximum length of $2^{31} - 1$ (2,147,483,647) bytes, or 2GB. (Note that this datatype is outdated, officially deprecated, 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.
    - `timestamp` (or `rowversion`): 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.

WOW! eBook
www.wowebook.org

    - `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 is 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.

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 whole numbers to fractional numbers, 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. While they have no error in the way they are stored, there can be issues when doing math with some of the types.

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 four 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:

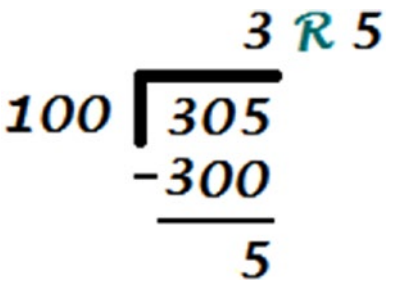
```
SELECT 1/2;
```


WOW! eBook
www.wowebook.org

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 returns integer values. The next intuitive leap you will probably make is that .5 should round up to 1, right? Nope, even the following query returns 0:

```
SELECT CAST(.99999999 AS integer);
```

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:



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 most users are likely desiring to get from a typical math equation, 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 Uses a singly byte 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 storing a number, other than `bit`.

WOW! eBook
www.wowebook.org

smallint

Domain: Whole numbers from `-32,768` through `32,767` (or -2^{15} through $2^{15} - 1$).

Storage: 2 bytes.

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 `int` 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 * 23) + (23 * 22) + (45 * 21) + (123 * 20)`. 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` falls down in the same manner with 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.

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.

WOW! eBook
www.wowebook.org

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 and 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` and `numeric` datatypes are 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. With SQL Server, these two are functionally the same, but they are not technically the same datatype.

```
SELECT name, system_type_id
FROM   sys.types
WHERE  name IN ('decimal','numeric');
```

Returns

```
name       system_type_id
---------- --------------
decimal    106
numeric    108
```

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.
- *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.

WOW! eBook
www.wowebook.org

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.

Row Compression Effect:

The value will be stored in the smallest number of bytes 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.

WOW! eBook
www.wowebook.org

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 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);
```


WOW! eBook
www.wowebook.org

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 round-off 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.

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.

WOW! eBook
www.wowebook.org

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 15-1.
- `real`
  - `real` is a synonym for `float(24)`.

Table 15-1. *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

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 numeric datatypes can deal with, but the range is enormous and is plenty for almost any scientific application. 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.OOOOOOOOOOOOOOOOOOOOOOOOOOOODDDDDDDDDDDDDD`
- `NNNNNNNNNNNNNNNOOOOOOOOOOOOOOOOO`

WOW! eBook
www.wowebook.org

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 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 was added that changed all of that. These datatypes are `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.

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).

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.

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:

WOW! eBook
www.wowebook.org

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. `datetime2(0)` is accurate to 1 second, `datetime(1)` to .1 seconds, up to `datetime(7)` as .0000001. The default is 7 or datetime2(7).

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` has.

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 bytes 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.

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.

WOW! eBook
www.wowebook.org

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).

Accuracy: 1 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 just the date, and possibly the 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 the time stored is midnight, 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. It is accurate to .003 seconds, leading to interesting round-off 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 heavily used 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 the time is midnight, 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.

WOW! eBook
www.wowebook.org

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 Time Zone Offset (`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. Execute 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);
```


WOW! eBook
www.wowebook.org

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 selection 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 to 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.

WOW! eBook
www.wowebook.org

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.

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 format or the ISO 8601 format 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('2013-01-01T14:23:00.120' AS datetime) AS withTime;
```


WOW! eBook
www.wowebook.org

For more information, check SQL Server Books Online under "Date and Time Format." Related to dates, a new function FORMAT was 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:

```
DECLARE @DateValue datetime2(3) = '2012-05-21 15:45:01.456'
SELECT @DateValue AS Unformatted,
       FORMAT(@DateValue,'yyyyMMdd') AS IsoUnseparated,
       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              IsoUnseparated 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 when building regionalized reports. Note that the Great Britain version doesn't list the day of the week, whereas the United States and France versions 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. Note that FORMAT can be slower than other techniques, so if it is being called millions of times, it could be a performance hit.

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 and Time 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:

WOW! eBook
www.wowebook.org

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 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 a query like the following, using the numbers table that we created back in Chapter 12 (and the code for which will be included in the Appendix B download):

```
SELECT I, CHAR(I)
FROM   Tools.Number
WHERE  I >=0 and I <= 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 presented 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.

Note that the length is optional. When used in a variable or column declaration, such as DECLARE @column char, the max length is 1. When used in a CAST or CONVERT, such as CAST(@column AS char), the max length is 30. Best practice is to always specify a length.

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 to keep your data behaving as is expected from char value.

---

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-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.

WOW! eBook
www.wowebook.org

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).

Note that the length is optional. When used in a variable or column declaration, such as DECLARE @column varchar, the max length is 1. When used in a CAST or CONVERT, such as CAST(@column AS varchar), the max length is 30. Best practice is to always specify a length.

Row Compression Effect:

No effect.

varchar(max)

Domain: ASCII characters, up to $2^{31} - 1$ characters (that is a maximum of 2GB 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 or 0 =: 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 or 1 =: You store varchar(max) values using 16-byte pointers to separate pages that just hold large objects. Use this setting if the varchar(max) data will only seldom be used in queries.

Discussion:

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, you likely 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 100MB LAN, sending 2GB is no instantaneous operation, for sure.

There are a couple things to look at:

- The UPDATE statement has a .WRITE() method 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);
```


WOW! eBook
www.wowebook.org

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)),8000)
                            + REPLICATE(CAST('X' AS varchar(max)),8000);
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 now have written that statement for many versions of this book). Worst of all, the text datatype is simply horrible to work with. Replace immediately 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 $2^{31} - 1$ characters (2GB 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.)

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 utilized 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.

WOW! eBook
www.wowebook.org

Just like the ASCII characters, you can see all of the different characters in the Unicode set by executing

```
SELECT I, NCHAR(I)
FROM   Tools.Number
WHERE  I >=0 and I <= 65535;
```

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 n 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% 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. Binary values are essential to the process of storing encrypted values in SQL Server. In Chapter 8, you'll learn about the encryption capabilities of SQL Server 2005.

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 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 hexadecimal literals such 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;
```


WOW! eBook
www.wowebook.org

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.

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.

A very common use for varbinary data is when you are using encryption of some sort. Encryption typically returns some binary value that is stored in binary when encrypted.

Row Compression Effect:

No effect.

varbinary(max)

Domain: Binary data, up to $2^{31} - 1$ bytes (up to 2GB 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 8.

WOW! eBook
www.wowebook.org

Storage: There are a couple of 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 always 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 spreadsheets 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 filestream and perhaps filetable 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 8 in the section on storing images and files.

Row Compression Effect:

No effect.

image

Just like the `text` datatype, the `image` datatype is deprecated. 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`

WOW! eBook
www.wowebook.org

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 much 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 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 give you a type conversion error.

Row Compression Effect:

Depends on the number of bits in the row. For a single bit value, 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.

WOW! eBook
www.wowebook.org

■ **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. While I doubt they ever change this, you should still change this column whenever you get the chance (not as urgently as `text` and `ntext`, but still…).

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 dbo.TestRowversion
(
   Value   varchar(20) NOT NULL,
   Auto_rv   rowversion NOT NULL
);

INSERT INTO dbo.TestRowversion (Value)
VALUES('Insert');

SELECT Value, Auto_rv
FROM dbo.testRowversion;

UPDATE dbo.TestRowversion
SET Value = 'First Update';

SELECT Value, Auto_rv
FROM dbo.TestRowversion;

UPDATE dbo.TestRowversion
SET Value = 'Last Update';

SELECT value, auto_rv
FROM dbo.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
Falue                Auto_rv
-------------------- ------------------
Last Update          0x00000000000007DC
```


WOW! eBook
www.wowebook.org

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 rowversions 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. I use rowversion columns in Chapter 11 to demonstrate optimistic locking.

Row Compression Effect:

Uses an integer structure 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, the chance that there will ever be any duplication in their values is tremendously remote, 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;
```

This returns (a similar value to) the following:

```
guidVar
------------------------------------------------------------
6C7119D5-D48F-475C-8B60-50D0C41B6EBFF
```

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 dbo.GuidPrimaryKey
(
   GuidPrimaryKeyId uniqueidentifier NOT NULL ROWGUIDCOL PRIMARY KEY 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:

WOW! eBook
www.wowebook.org

```
INSERT INTO dbo.GuidPrimaryKey(Value)
VALUES ('Test');
```

Then run the following command to view the data entered:

```
SELECT *
FROM   dbo.GuidPrimaryKey;
```

This returns the following result (though your key value will be different or this would not be a GLOBALLY unique value, would it?):

```
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.

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 dbo.GuidPrimaryKey;
GO
CREATE TABLE dbo.GuidPrimaryKey
(
   GuidPrimaryKeyId uniqueidentifier NOT NULL
                    ROWGUIDCOL DEFAULT NEWSEQUENTIALID()
                    CONSTRAINT PKGuidPrimaryKey PRIMARY KEY,
   Value varchar(10)
);
GO
INSERT INTO dbo.GuidPrimaryKey(value)
VALUES('Test'),
      ('Test1'),
      ('Test2');
GO

SELECT *
FROM   GuidPrimaryKey;
```


WOW! eBook
www.wowebook.org

This returns something like the following, with a notable progression to the values of the `GuidPrimaryKeyId` column values:

```
GuidPrimaryKeyId                     Value
------------------------------------ ----------
18812704-49E3-E011-89D1-000C29992276 Test
19812704-49E3-E011-89D1-000C29992276 Test1
1A812704-49E3-E011-89D1-000C29992276 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. The values are not guaranteed to remain sequential with existing data in the table after a Windows reboot.

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 very likely to be unique is a major plus, rather than requiring you to generate them in a single-threaded manner to ensure uniqueness. (One of the tech editors for the book reported that he sees them occasionally in a highly concurrent banking system that relies on GUID values for surrogate keys…it is not impossible to have collisions.)

Row Compression Effect:

No effect.

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:

Not applicable.

table

The `table` type is kind of two different things. 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, 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

WOW! eBook
www.wowebook.org

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
```

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 dbo.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');
```


WOW! eBook
www.wowebook.org

This returns the following result:

```
Value
--------------------------------
testValue
```

One interesting thing about the `table` datatype is that the tables that are created as variables aren't subject to transactions, since they are variables. 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
```

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 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 6, 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 have used several times is a 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:

```
USE WideWorldImporters;
GO
CREATE TYPE GenericIdList AS TABLE
(
```


WOW! eBook
www.wowebook.org

```
    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 @PeopleIdList GenericIdList;
INSERT INTO @PeopleIdList
VALUES (1),(2),(3),(4);

SELECT PersonId, FullName
FROM   Application.People
         JOIN @PeopleIdList AS list
            on People.PersonId = List.Id;
```

This returns the following:

```
PersonId    FullName
----------- --------------------------------------------------
1           Data Conversion Only
2           Kayla Woodcock
3           Hudson Onslow
4           Isabella Rupp
```

And as of 2014, you can use a memory-optimized table type as well:

```
--database must support in-memory with in-mem filegroup
CREATE TYPE GenericIdList_InMem AS TABLE
(
    Id Int PRIMARY KEY NONCLUSTERED --Use nonclustered here,
                                        --as it should be fine for
                                                          --typical uses

) WITH (MEMORY_OPTIMIZED = ON);
```

And it will execute the same way, though it can be a good bit faster in some scenarios:

```
DECLARE @PeopleIdList GenericIdList_InMem;
INSERT INTO @PeopleIdList
VALUES (2),(3),(4);

SELECT PersonId, FullName
FROM   Application.People
         JOIN @PeopleIdList AS list
            ON People.PersonId = List.Id;
```

Of course, you can then use either of the types in your stored procedure creation statements as well:

```
CREATE PROCEDURE Application.People$List
(
```


WOW! eBook
www.wowebook.org

```
    @PeopleIdList GenericIdList READONLY
)
AS
SELECT PersonId, FullName
FROM   Application.People
         JOIN @PeopleIdList AS List
            ON People.PersonId = List.Id;
```

Unfortunately, you cannot pass a set of row constructors to the stored procedure; you will need to declare and load a table variable to use this construct from T-SQL:

```
DECLARE @PeopleIdList GenericIdList;


INSERT INTO @PeopleIdList
VALUES (2),(3),(4);

EXEC Application.People$List @PeopleIdList;
```

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 technique 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 allows 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.

WOW! eBook
www.wowebook.org

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 NULLs differently from NULLs in other datatypes.
- *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_variants and the datatypes don't match, then the values won't match 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          Latin1_General_100_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
```


WOW! eBook
www.wowebook.org

Not Simply Scalar Datatypes

This section deals 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. This was briefly touched upon in Chapter 8.
- *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.

WOW! eBook
www.wowebook.org

Index

A

Ad hoc SQL
  advantages, 661, 698
  batching interface, 674
  code-wrapping mechanism, 673
  disadvantages, 698
  functional code, transaction, 673
  low cohesion and high coupling, 671, 673
  parameterization
    ALTER DATABASE command, 669
    encapsulation layer, 671
    LIKE condition, 669
    literal value, 669
    sp_executeSQL, 670
    sp_unprepare and sp_prepare statement, 671
    tables, 668
  performance tuning, 677–678
  runtime control
    contact table, 663
    database creation, 661–662
    data changed property, 665
    FROM clause, 661
    IF blocks, 667
    SELECT clause, 661
    UPDATE statements, 665
    varchar(max) columns, 665
    WHERE clause, 666
  security issues, 675
  shared execution plans, 667–668
  SQL injection, 676–677
  SQL Server Management Studio, 661
AFTER triggers, 288
  cascading inserts, 297
  child to parent cascading, 300
  databases and servers relationship
    child insert and update, 305
    parent delete, 304
    parent update, 304
  INSTEAD OF triggers, 307–308
  range checks, multiple rows
    error message, 292–293
    FROM clause, 291
    trigger events, 293
    WHERE clause, 291
  summary value maintenance
    DELETE trigger, 297
    EXISTS filter, 294
  types, 288
Aggregation reporting style, 706–707
Alternate key (AK), 20, 66–67
Analysis paralysis, 135
Analytical reporting style
  Bill Inmon approach, 706
  denormalization, 706
  dimensional modeling, 706
    business process, 708, 710
    datedimension (*see* Date dimension)
    health care payer dimensional model, 709
    slowly changing dimension, 713–717
    snapshot fact, 724–727
    snowflake dimension, 717–719
    tables types, 708
    transactionfact (*see* Transaction fact)
    type dimension, 719–721
  Ralph Kimball's approach, 706
  summary modeling
    additional summary tables, 733
    benefits, 730
    initial summary table, 731–732
Anti-patterns
  cached object, 403
  data in query, 402
  domain tables, 401
  domain values, 400
  expandability and control, 403
  foreign key constraints, 403
  generic key references, 398
    GUID key, 404
    multiple tables, same key, 404
    objects, maximum usability/flexibility, 405

© Louis Davidson 2016
L. Davidson and J. Moss, *Pro SQL Server Relational Database Design and Implementation*,
DOI 10.1007/978-1-4842-1973-7

WOW! eBook
www.wowebook.org

Anti-patterns (*cont.*)
multiuse domain table, 401
no datatype standardization, 399
normalization process, 401
one domain table per purpose, 402
one-size-fits-all domain, 398
overusing unstructured data, 399
performance considerations, 403
poor domain choices, 399
poor normalization practices, 399
relational databases, 400
undecipherable data, 398–399
unstructured data, 406
Application roles, 439–442
Approximate uniqueness, 339
Architectural buildingblocks. *See* Data access strategies
Ascynchronous contention
handling timing issues, 616
locks, 616
logical unit, 622–624
row-based optimistic locking, 617
row-level change detection, 621–622
validation columns, 618–621
Attributes
alternate key, 66–67
foreign key, 68
naming, 63–64, 94
primary key, 64–66

B

Bill Inmon approach, 706
Binary relationships, 27
Bottom-up approach, 706
Boyce-Codd Normal Form (BCNF), 158
definition, 158–160
multiple columns with same prefix, 167
repeating groups of data, 168
summary data, 169
B-tree indexes, 494–495
Built-in database roles, 434
Bulk uniqueness
data loading, 329
grouping sets, 332
inventory and utilization storage, 326
inventory model, 327
row constructor syntax, 330
rows and data manipulation, 326
table creation, 328
table implementation, 329
Business rules identification, 132–133
Bw-Tree index, 530–531

C

Cadidate key
composite key, 19
definition, 19
natural keys, 21–22
surrogate keys, 24–25
types, 20
Calendar table
application, 642
business calendar, 642
creation, 638
FiscalYear, 644
floating windows, 642, 645
functions, 639
generic events/time ranges, 642
grouping types, 640
Relative____Count columns, 645
set of values, 641–642
SQL Server date functions, 638
CHARINDEX function, 214
Check constraints
ALTER TABLE statement, 276, 278
catalog number mask, 275
complex expressions, 273
data seeding, 275
declarative constraints, 273
error handling, 273
entering invalid value, 287
error description, 285
ErrorMap table, 286
error message, 285, 287
mapping table, 285
rudimentary error-mapping scheme, 285
TRY-CATCH error handling, 285
example schema, 274
INSERT statement, 276
release-date column, 275
simple expressions, 273
date range checks, 279
empty strings, 279
LEN function, 280
trusted status and values, 280
value reasonableness, 279
sys.check_constraints catalog object, 277
sys.check_constraints query, 279
UPDATE statement, 277
using functions
Boolean expression, 281–282
CLR-based objects, 282
complex scalar validations, 282
DML modification statement, 282
scalar T-SQL function, 281
single-row validation code, 284
tables access validation, 282

WOW! eBook
www.wowebook.org

T-SQL function, 282
UDFs, 282
user-defined functions, 284
WITH CHECK, 277
WITH NOCHECK setting, 276
CHECK_EXPIRATION, 417
CHECK_POLICY, 417
Chen Entity Relationship Model (ERD), 90, 92–93
Chunked updates, 215
Codd's rule, RDBMS
comprehensive data sublanguage, 5
distribution independence, 8
dynamic online catalog, 5
guaranteed access, 4
high-level insert, update and delete, 6
information principle, 3
innovation, 4
integrity independence, 7
logical data independence, 7
NULL values, 4
physical data independence, 6
rows and columns, 3
updating rules, 6
Code-based denormalizations, 230
Cohesion, 671
Collation, 220–222
Column-level security, 432–433
Columnstore indexes, 536
Common Language Runtime (CLR), 701, 703
complex procedural logic and computations, 701
extended stored procedures, 702
.NET framework classes, 701
objects and functionality, 702
rich language support, 701
string manipulation, complex statistical calculations and custom encryption, 701
Visual Studio, 702
Common Table Expression (CTE), 333
Compression, 546–547
Conceptual model
entities
audit trails, 106
events, 106
records and journals, 107
identifying entities
document, 104
entity listing, 108
groups, 105
ideas, 103
objects, 102–103
persons, 100–101
places, 101
relationships identification, entities
listing relationships, 116–118
many-to-many relationships, 114
one-to-N relationships, 109–114
testing, 119–120
Concurrency coding
isolating sessions
lock characteristics, 581
lock modes, 582–585
lock types, 581
lost update, 580
logical unit, 622
MARS, 562
multiuser database, 561
OS and hardware, 562
parallelism, 561
Resource Governor, 560
row-based locking
add a time column method, 617
check all columns method, 617
optimistic lock columns, 618
row-level optimistic lock coding, 621
timestamp column method, 617
tradeoffs, 560
transactions (*see* Transactions)
Constraints keys, 19–20
Contained database authentication, 421
Contained database (CDB), 416
ContainedDBSecurityExample, 424
Controlling access
asCaller variant, 465
asDbo variant, 465
catalog views, 460
crossing database lines, 464
dbo.testDboRights procedure, 463
dbo user, 464
db_owner's role, 463
EXECUTE AS statement, 458, 462
primary schema, 459
row-level security, 464
security context, 458
SELECT permissions, 460
stored procedures and scalar functions, 456–458
table/object-level security, 455
views and table-valued functions, 456, 465–466
CREATE DATABASE statement, 421
CREATE SEQUENCE dbo.test, 234
Cross-database chaining
ALTER AUTHORIZATION DDL statement, 469
ALTER DATABASE, 469
Cross DB Ownership Chaining, 469
EXECUTE AS, 471
OLTP databases, 468
server principal, 472
stored procedures, 471
sys.databases catalog view, 471

WOW! eBook
www.wowebook.org

Cross-database chaining (*cont.*)
sys.dm_db_uncontained_entities dynamic management view, 472
uncontained users, 473
Windows Authentication, 469
Cross DB Ownership Chaining, 469
Crossing database lines
backups, 467
certificate-based trust, 475–477
contained databases, 467
developer tools, 467
distributed queries, 477–478
foreign key constraints, 467
impersonation, 474–475
off-the shelf package, 468
ownership chain, 468

D

Data access strategies
ad hoc SQL, 657
connecting to the server, 413, 416
database model
Azure DB databases, 421
Connection Properties, 422–423
contained user, 421–422
CREATE DATABASE statement, 421
object explorer, SSMS, 424
security context, 422
sp_configure, 421
SQL Server 2012, 425
SQL Server Authentication, 421
SSMS, 422
uncontained database, 424
Windows Authentication, 422
Entity Framework, 658
host server security configuration, 413–414
impersonation, 413, 425–427
interface creation, 658
login and database user, 417–418
non-data tier rule, 658
object-relational mapping tools, 658
principals and securables, 413
stored procedures, 702
advantages and disadvantages, 699
complex plan parameterization, 687–688
cross-platform coding, 694
data format, 700
data-manipulation code, 700
encapsulation, 680–682
EXECUTE AS clause, 700
fine-tuning without program changes, 688–689
high initial effort, 694
injection attack value, 682
in-memory engine, 689
INSTEAD OF trigger, 697, 698
JOIN clause, 683
mutable business logic and rules, 700
optional parameters, 694
precompiled stored procedures, 682
QUOTENAME() function, 683
RETURN statement, 678
rows retrieval, 679
security, 680, 685–687
sp_executeSQL and parameterization, 683
structure, 678
WHERE clause, 683–684
T-SQL, 657–658
CLR (*see* Common Language Runtime (CLR))
DBAs, 701
flow language, 702
.NET language, 701
Database components
application, 655
Calendar table, 628
database constructs, 627
database design process, 627
logging objects, 628
numbers table, 627
reference/demographic information, 655
security, 655
SQL Server, 655
third-party system, 628
universal models, 627
utility objects, 628
SQL Server 2016 WideWorldImporters database, 628
Database object securables
application roles, 439–442
built-in database roles, 435
column-level security, 432–433
grantable permissions, 429–430
principals, 428
roles, 434
schemas, 442–443
table security, 430–432
user-defined database roles, 434–439
users permissions, 429
Database principals, 415
Database scoped, 415
Database security
application layer control security, 413
auditing, 412
auditing SQL server usage, 483–484
Azure SQL Database, 413
controlling access to data via T-SQL coded objects, 412
crossing database lines, 412

WOW! eBook
www.wowebook.org

database access, 412
database design and implementation, 412
database object securables, 412
audit specification, 484–485
definition, 484–485
enabling, 485
host server, 412
instance/SQL server, 412
obfuscating data, 412
privacy policy, 411
row-level security, 412
storing data, 488–489
viewing, audit configuration, 487
viewing, audit trail, 486–487
Data definition language (DDL), 10
basic check constraints
user-defined functions, 250
[WITH CHECK | WITH NOCHECK] specification, 244
basic table structures
CREATE TABLE statement, 227–228
check constraints
Boolean expression, 250, 251
MessageTopic table, 251
NULL, 250
triggers, maintain automatic values, 252–255
T-SQL, 250
validation routines, 249
ConferenceMessaging database, 225
database documentation
Descriptions in Management Studio, 258
fn_listExtendedProperty object, 258
functions and procedures, 256
parameters, 256
reindexing schemes, 256
repository information, 255
script, 257
tables, column, and schemas limitations, 259
data modeling tools, 224
default constraints, 242–243
foreign keys
addition syntax, 244
ALTER TABLE statement, 243
CASCADE operations, 247
CREATE TABLE statement, 243
cross-database relationships, 249
parent-to-child relationship, 245
SQL Azure, 249
surrogate keys, 244
metadata
catalog view, 263
constraints, 261
INFORMATION_SCHEMA views, 260, 263
schema list, 259
sys schema objects, 259
table and column name, 260
triggers, 262
table structures
columns and base datatypes, 229–231
non-natural primarykeys (*see* Non-natural primary keys)
nullability, 231
schema, 228–229
uniqueness constraints addition
alternate key constraints, 240–241
indexes, 238, 241
primary key constraints, 238–239
Data gathering, 41
artifacts, 46
client interviews
communication gap, 46
formal, 46
one-to-one sessions, 47
structured database, 46
data integration with system, 51
data location, 50
data type needed, 47, 48
data worth, 51
documentation
audit plans, 54
contracts/client work order, 53
RFQ and RFP, 53
service level agreement, 53
E-R modeling method, 45
governing rules, 48
graphical model, 45
multiuser data, 51
prototypes, 54
reported data, 49
requirements, 44
subject area, 45
systems and prototypes, 52
UML, 45
Data modeling
defined, 58
descriptive information, 88–89
IDEF1X, 58–59
Data protection
checkconstraints (*see* Check constraints)
concurrency, 272
custom front-end tools, 272
DMLtriggers (*see* DML triggers)
generic data manipulation tools, 272
raw queries, 272
reliability and integrity, 271
routines, 272
UNIQUE constraint, 271

WOW! eBook
www.wowebook.org

Datatype configurations
Boolean/logical values, 212–213
complex CLR datatypes, 216
large-value datatype columns, 214–215
user defined type/alias, 215–216
Date dimension
entity representation, 711
IDENTITY column, 711
standard attributes, 711
stored procedure, 712–713
table creation, 711
unknown rows, 712
Deadlock, 584–585
Delimited identifiers, 192
Denormalization, 178–180
DENY, 415
Dependencies, 2
Descriptive information, 88–89
Distribution Independence, 8
DML triggers
advantages, 288
AFTER triggers, 288
cascading inserts, 297
child to parent cascading, 300
databases and servers relationship, 303
summary value maintenance, 293
types, 288
and constraints errors
constraint mapping function, 318
doomed transaction, 317
ErrorHandling.ErrorLog$insert object, 318
ROLLBACK, 314
SELECT statement, 315
THROW, 318
transaction state, 315
transaction without rolling back, 316
TRY-CATCH block, 313
INSTEAD OF triggers, 288
no action on table, 311
redirecting invalid data, 308
OLTP operations, 287
Domain implementation, 17–18, 94
collation selection, 220–222
column/table, 207–209
consistency, 205
datatype selection
approximate numeric data, 210
binary data, 210
boolean/logical values, 212–213
character/string data, 70, 211
complex datatypes, 216
date and time, 210
deprecated/bad choice types, 212
hierarchyId, 211
large-value datatype columns, 214–215
messaging system model, 219
precise numeric data, 209
rowversion, 211
spatial types, 211
sql_variant, 211
uniqueidentifier, 211
user defined type/alias, 215–216
varchar(max) column, 217
varchar(N), 216
XML, 211
defining common domains, 69
documentation, 205
ease of implementation, 205
implementation phase, 71
logical modeling, 71
nullability selection, 219–220
reusable template attributes, 70
subclasses, 69
Dynamic management view (DMVs), 418
fragmentation, 554
index utilization statistics, 553–554
missing indexes, 550–553
Dynamic online catalog, 5

E

Encapsulation, 680–681
Entire key dependency, 162–163
Entities
database objects naming, 60–61
dependent entity and independent entity, 59
naming, 94
Entity-attribute-value (EAV)
dynamic statement, 390
property schema, 387
property table, 387
row insertion, 388
sql_variant column format, 391
trappable error, 389
Entity-relationship (E-R) modeling, 45
EXECUTE AS statement, 413, 425, 440
EXECUTE AS LOGIN, 427

F

Fifth normal form
data redundancy, 175
dependencies, 175
intercolumn dependencies, 175
interpretations, 177
Filestream, 215, 371
Floating point, 754
Foreign key, 68
Fragmentation, 501
Fundamental processes identification, 133–134

WOW! eBook
www.wowebook.org

G

Game tables, 300
Generalization
  database implementation, 381
  extended description, 385
  table creation, 382
  table update, 383
GETDATE() function, 252–253
Globally unique identifier (GUID), 24
GRANT, 415
Guaranteed access, 4

H

Hash indexes, 531–533
Heap, 509–510
Hierarchies, 356
  hierarchyTypeId type, 364
  query optimizations
    Kimball helper table, 368, 370
    nested sets, 369
    path technique, 368
  self-referencing relationship
    graphs, 361
    trees, 357

I

Identifiers, 121–123
IDENTITY property, 232–233
Indexes
  clustered indexes
    ALTER INDEX REORGANIZE, 504
    clustered index scan, 504
    clustered tables, 502–503
    clustering key, 502–503
    GUID, 503
    NEWSEQUENTIALID() function, 503
    OLTP setting, 504
    range queries, 504
    SET STATISTICS IO, 506
  dynamic management view queries
    fragmentation, 554
    index utilization statistics, 553
    missing indexes, 550
  foreign keys
    domain tables, 540
    indexed views, 543
    many-to-many resolution table relationships, 542
    OLTP database, 538
    one-to-one relationships, 542
    ownership relationships, 541
    sample relationship, 538
    trigger, 539
  memory-optimized (*see* In-memory OLTP tables)
  multiple columns
    composite indexes, 519–521
    covering indexes, 522–523
    index keys, sort order, 524
    multiple indexes, 523–524
  nonclustered indexes
    abstract representation, 507
    clustered table, 508–509
    DBCC SHOW_STATISTICS command, 515
    DELETE operation, 513
    dynamic management views, 512
    filtered index, 518
    heap, 510, 525
    histogram, 516–518
    INSERT operation, 512
    leaf page, 507
    profiler, 512
    queries density, 516
    row locator, 507
    SELECT operation, 512
    slow queries, 512
    TABLESAMPLE clause, 518
    UPDATE operation, 512
    user_scans, 513–514
  OLTP patterns
    ALTER INDEX REORGANIZE, 538
    OLTP setting, 538
    primary key, 537
  SET SHOWPLAN_TEXT commands, 492
  structure, 494–496
  unique indexes, 526
Information engineering (IE), 90–92
In-memory engine, 659–660, 689
In-memory model, 189–190
In-memory OLTP tables
  Bw-tree index, 530–531
  hash index, 531–533
  index pointers, 529
  index stats, 555–556
  record header, 528
  row structure, 528
  STATISTIC TIME, 535
  table structure, 529
  timestamp, 528
INSTEAD OF triggers
  no action on table, 311
  redirecting invalid data, 308
Integration Definition for Information Modeling (IDEF1X), 58
Integrity independence, 7

WOW! eBook
www.wowebook.org

Isolation levels
deadlock, 584–585
FROM clause, 583
lock compatibility, 583
lock modes, 582
lock types, 581–582
pessimistic concurrency enforcement, 580
READ COMMITTED, 579, 588–589
READ COMMITTED SNAPSHOT, 602
READ UNCOMMITTED, 586–587
REPEATABLE READ, 589–590, 610–611
SELECT statement, 578
SERIALIZABLE, 590–591, 611, 613
SNAPSHOT, 579, 598, 608–609
syntax, 579
sys.dm_exec_sessions, 579

J

JSON-formatted values, 14

K

Key implementation
alternate keys
constraints, 203
logical model, 202
primary key
existing columns, 197
new surrogate value, 198–200, 202
Kimball helper table, 370

L

Locators, 124–126
Locks
application locks, 593–597
and foreign keys, 591, 592
Logging objects, 652–654
Logical data independence, 7
Logical model
client review, 137
completed patient entity, 136
discovery phase, 136
documentation, 135
identifying attributes and domains
descriptive information, 123
identifiers, 121–123
locators, 124–126
relationship attributes, 127
values, 126
identifying business rules, 132–133
identifying fundamental processes, 133–134
testing, 137

M

Many-to-many
relationships, 85–86
Metadata, 18
Missing values (NULLs), 15–16
Modeling language, 94
Multiparent hierarchies, 361
Multiple active result sets (MARS), 562
Multi-Value Concurrency Control, 597
Multivalued
dependency (MVD), 172

N

Nested sets, 369
Nonbinary relationships, 31
Nonnatural primary keys
DDL, tables building, 236–237
default constraint, 233–236
IDENTITY property, 232–233
manual management, 232
Nonsubversion rule, 8
Normal Form Recap, 181
Normalization
categories, 142
guiding principles, 180
normal forms, 141
process, 142
relationships between columns
BCNF, 158–160, 167–169
dependency between rows, 166
entire key dependency, 162–163
partial key dependency, 160–161
positional meaning, 170
surrogate keys effect, 163–165
tables with multiple meanings
fifth normal form, 175–177
fourth normal form, multivalued dependencies, 172–174
Nullability, 219–220
NULL values, 4
Numbers table
attributes, 629
Erland Sommarskog's web site, 630
EXECUTE and SELECT, 631
integers, 631
non-negative integers, 628, 629
SELECT clause multiplier, 630
separating comma-delimited items, 635–637
sequence of numbers, 634
string, 632–634
usage, 632

WOW! eBook
www.wowebook.org

O

Obfuscating data
  database and users, 478
  dynamic data masking, 478, 480–482
  encrypting data, 478, 479
  sys_admin server role, 478
Object relational mapping (ORM) tools, 201
On-disk model, 191
One-to-many relationships, 27–29
One-to-N relationships
  multivalued attributes and domains, 112
  types, 110
Operational querying, 740–743
Operational reporting modeling
  benefits, 736
  data model, 737
  in-memory OLTP, 737–739
  reporting queries, 736
Optimistic concurrency enforcement
  connections
    inserts, 614
    uniqueness violation, 615
  foreign keys, 615–616
  in-memory OLTP tables
    filegroup, 606
    REPEATABLE READ, 607
    SERIALIZABLE, 608
    SNAPSHOT, 605, 608
    timestamp, 604–605
  on-disk
    READ COMMITTED SNAPSHOT, 602–604
    SNAPSHOT isolation, 598–601
OS and hardware, 562

P, Q

Parallelism, 561
Parameter sniffing, 687
Partial key dependency, 160–161
Partitioning, 548–550
Patterns
  data-driven design, 322, 340, 362
  generalization, 322
  hierarchies, 322
  historical/temporal data, 322, 341–342
  images and documents, 322
    API, 377
    binary data, 371
    binary format data storage, 378
    binary storage, SQL Server's storage engine, 371
    CREATE TABLE TestSimpleFileStream, 373
    DIRECTORY_NAME parameter, 374
    encryption, 377
    files location, 377
    filestream access, 372
    filestream attribute, 374
    filestream files, 373
    filetable directory, 376
    FILETABLE_DIRECTORY, 374
    FileTableRootPath() function, 375
    filetable style, 374
    generalization, 380
    image and data backup, 377
    MSSQLSERVER, 374
    NON_TRANSACTED_ACCESS, 374
    path reference, file data, 371
    Remote Blob Store API, 376
    ROWGUIDCOL property, 373
    security, 377
    size, 377
    stream_id, 375, 379
    transactional integrity, 371, 377, 379
    user-specified datastorage (*see* User-specified data storage)
    utilization, 377
    varbinary(max) column, 371
  storing user-specified data, 322
  uniqueness, 322, 323, 361, 408
Physical database structure
  allocation map, 497
  bulk changed map, 498
  data pages, 498
  data row, 500
  dictionary compression, 547
  differential changed map, 498
  files and filegroups
    default filegroup, 496
    sys.filegroups catalog view, 497
  index allocation, 498
  index data, 497
  mixed extent, 498
  overflow data, 497
  overflow pages, 498–499
  page compression, 546
  page free space, 498
  page split, 500–501
  partitioning, 548–550
  prefix compression, 547
  row compression, 546
  table data, 497
  uniform extent, 498
Physical model implementation
  adding implementation columns, 223–224
  database generation tools, 268
  database model, 188
  database requirements, 187
  datatype and nullability, 269
  DDL (*see* Data definition language (DDL))

WOW! eBook
www.wowebook.org

Physical model implementation (*cont.*)
Developer Edition SQL Server, 187
document and script, 269
domainimplementation (*see* Domain implementation)
foreign key constraints, 269
in-memory OLTP, 189–190
key implementationselection (*see* Key implementation)
name selection
delimited identifiers, 192
identifiers, 191
model name adjustments, 195–196
naming columns, 194–195
naming tables, 192–193
policy-based management, 192
regular identifiers, 191
sysname, 191
naming objects, 268
normalization maintanance, 268
on-disk, 189–190
procedure, 186
schemas, 222–223
structure testing
CATCH block, 266
delete statements, 264
foreign key constraints, 263
integration testing, 264
non-alphanumeric character, 265
RAISERROR statement, 264
test scripts, 263
user defined topic, 267
user handle, 265
tables and columns documentation, 188–189
template domains, 269
test script, 269
UNIQUE constraint, 269
Policy-based management, 192
Precise numeric data
decimal values
float and real datatypes, 750
money types, 752
SET NUMERIC_ROUNDABORT ON, 751
testvar decimal, 751
integer values
bigint, 749
int, 749
smallint, 749
tinyint, 748
Primary key (PK), 20, 64–66
existing columns, 197
new surrogate value
advantages, 198
constraint, 199
disadvantages, 198
IDENTITY property, 199
ORM tools, 201
single-column key, 198, 201

R

Ralph Kimball's approach, 706
Range uniqueness
appointment, 334, 337
data testing qurey, 335
delete operation, 335
row removal, 335
table creation, 334
Recursive relationship, 80–82
Regular identifiers, 191
Relational database design, 1
binary relationship, 27
cardinality, 26
Codd's rule, RDBMS, 3
nonsubversion rule, 8
database-specific project phase, 34
conceptual, 35–36
fundamental techniques, 35
logical, 35–36
physical, 35, 37
storage, 35, 37
dependency
determinants, 33
functional, 32–33
design phases, 2
entity relationship, 2, 26
foreign key, 25
history, 2–3
many-to-many relationship, 30
nonbinary relationship, 31
one-to-exactly N relationship, 30
one-to-many relationship, 27–28
parent and child tables, 26
relational programming, 2–33
SQL standards, 8
structure recognition, 9
databases and schemas, 10
domain definition, 17
key (*see* Candidate key)
metadata storage, 18
missing (NULL) values, 15–16
tables, rows and columns. Tables, rows and columns
structures, 2
Relational programming, 33–34
Relationships, 94
cardinality, 78–80
documentation, 116–117
entity, child and parent, 72
identifying, 73–74

WOW! eBook
www.wowebook.org

many-to-many, 85–86
nonidentifying, 74–75
one-to-many, 76
optional relationships, 75, 76
recursive, 80–82
role name, 76–78
subtypes, 82–84
types, 72
verb phrases, 86–88
Reporting design
aggregation querying
indexing, 735–736
queries, 734–736
analytical querying
indexing, 728, 730
queries, 727–728
requirements-gathering process, 707
styles (*see* Reporting styles)
Reporting styles
aggregation reporting, 706–707
analyticalreporting (*see* Analytical reporting style)
innies and outties, definition, 705
operational querying
indexing, 740–743
queries, 740
operational reporting, 707
Request for proposal (RFP), 53
Request for quote (RFQ), 53
Requirements-gathering process, 708
Resource Governor, 560
Reverse cascade operation, 302
REVOKE, 415
RoundedMessageTime, 218–219
RowCreateTime column, 242
RowLastUpdateTime, 242, 252
Row-level security
BLOCK predicate, 452
controlling security, 445
CREATE SECURITY POLICY statement, 449
database principal access, 445
data-driven, 454–455
features, 444
FILTER predicate, 452
filter works, 451
ownership chaining, 445
policy, 450
schema, 448
schema-bound view, 448
security function, 449
side-channel ways, 448
simple table-valued function, 450
single user, 454
specific-purpose views, 446–448
SQL Server 444, 2016
user-defined function, 448

S

Scalar datatype
approximate numeric data, 745, 754
binary data, 746
binary length, 765
image, 767
varbinary length, 766
varbinary max, 766
bit, 768
character data, 746
character strings
char length, 761
nchar, nvarchar, ntext, 764
text, 764
varchar length, 762–763
varchar max, 763
cursor, 772
date and time data, 745
date, 755
date functions, 758
date ranges, 759
date representation, text formats, 760
datetime, 757
datetime2, 756
datetimeoffset, 756
smalldatetime, 757
time, 755
non-scalar types, 778
precise numeric data, 745
rowversion, 768
sql_variant, 776
table valued parameters, 774
table variables, 772
uniqueidentifier, 770
Schema, 222–223, 415
SCOPE_IDENTITY(), 252, 254
Second Normal Form, 158
SELECT * FROM <tablename>, 6
Selective uniqueness
EXISTS query, 326
filtered indexes, 323
indexed view, 323
NULL, 325
row creation, 324
Server scoped, 415
Single-parent hierarchies
adjacency list, 358
children position, 360
circular references, 361
company hierarchy, 357

WOW! eBook
www.wowebook.org

Single-parent hierarchies (*cont.*)
functional language, 357
procedural language, 357
query code, children, 359
recursive type query, 361
table creation, 358
tree structure searched depth first, 357
tree structure with levels, 358
Slowly changing dimension
isCurrent flag, 715
member dimension table, 714
preferred query, 716
provider dimension table, 716–717
rejected query, 716
start date and end date, 716
types, 713
Smart keys, 23
Snapshot fact, 724–727
Snowflake dimension, 717–719
sp_setapprole procedure, 441
SQL Server 2012, 416
SQL Server Authentication login, 426
SQL Server Management Studio, 661
SQL Server principals, 415
SQL Standards, 8–9
Stored procedures
advantages, 679–680
complex plan parameterization, 687–688
dynamic procedures, 682–684
encapsulation, 680–681
fine-tune without program changes, 688–689
high initial effort, 694
in-memory engine, 690–693
INSTEAD OF trigger, 697–698
optional parameters, 694–695
parameter sniffing, 687
RETURN statement, 678
rows retrieval, 679
sales.contact table, 695
security, 685–687
structure, 678
Structure English Query Language (SEQUEL), 8
Sublanguage, 5
SUBSTRING function, 214
Subtypes relationship, 82–84
Surrogate keys effect, 218
tables chained, key migration, 165
Sysname, 191

T

Table and column shape
atomicity, complex datatypes, 144
differentiate between rows, 155–156
first normal form, 143
column names with numbers, 157
data characteristics, 157
e-mail addresses, 144–146
interpretations, 152
names, 147–148
separator-type characters, 157
tables, defined keys, 158
telephone numbers, 149–151
rules sets forth, 143
same number of values, 152–154
Table security, 430–432
Tables, rows and columns
atomic values, 11
basic data representation terms, 13
column term breakdown, 14
record manager, 12
relational theory, 12
row term breakdown, 15
scalar value, 11
Temporal extensions
configuration, 346–349
limitations, 345
multiple row, 352–353
setting/rewriting history, 354–356
single row, 349–352
Third Normal Form, 158
Transactions
atomicity and consistency, 564
autonomous, 570
BEGIN TRANSACTION, 565
bulk logged model, 564
claim payment fact table, 721, 723
COMMIT TRANSACTION, 565
definition, 563
degenerate dimensions, 724
distributed, 572–573
DML and DDL statement, 564
error handling, 574–576
explicit *vs.* implicit, 576–577
foreign keys, 722
full log model, 564
full recovery model, 566
isolation and durability, 564
nested, 568–570
preferred query, 723
rejected query, 723
RESTORE DATABASE command, 567
ROLLBACK TRANSACTION, 565–566
savepoints, 571–572
simple log model, 564
states, 573
syntax, 565
Transact SQL (T-SQL), 141
Trigger. *See also* AFTER triggers; INSTEAD OF triggers

WOW! eBook
www.wowebook.org

constraint mapping function, 318
doomed transaction, 317
ErrorHandling .ErrorLog$insert object, 318
ROLLBACK, 314
schema creation, 342
SELECT statement, 315
THROW, 318
transaction state, 315
transaction without rolling back, 316
TRY-CATCH block, 313
update table, 344
Type dimension, 719–721

U

Unified Modeling Language (UML), 45
uniqueidentifier columns, 771
Uniqueness, patterns
approximate, 339
bulk
data loading, 329
grouping sets, 332
inventory and utilization storage, 326
inventory model, 327
pieces, 331–333
row constructor syntax, 330
rows and data manipulation, 326
table creation, 328
table implementation, 329
issues, 323
range
appointment, 334, 337
data testing qurey, 335
delete operation, 335
row removal, 335
table creation, 334
selective
EXISTS query, 326
filtered indexes, 323
indexed view, 323
NULL, 325
policy numbers, 323
row creation, 324
two rows with null, 324
User-defined database roles, 434–439
User-defined functions, 273
User interface (UI), 41
User-specified data storage
adding columns to table
ALTER TABLE, 392
CHECK constraint, 395
column set, 393, 396
sparse columns, 396
EAV
dynamic statement, 390
EquipmentPropertyType table creation, 387
property schema, 387
property table, 387
row insertion, 388
sql_variant column format, 391
trappable error, 389
Equipment table, 386
methods, 386
T-SQL flexibility, 385
Utility objects
ad hoc manner, 647
database, 648
extended DDL utilities, 650–652
monitoring objects, 648–649
types, 647–648

V

Varbinary(max), 215, 216
Varchar(max), 214, 215
Verb phrases, 86–88

W, X, Y, Z

Waterfall method, 42
Windows Authentication, 417, 439
Windows principal, 415, 417
WorldWideImporters database, 646

WOW! eBook
www.wowebook.org