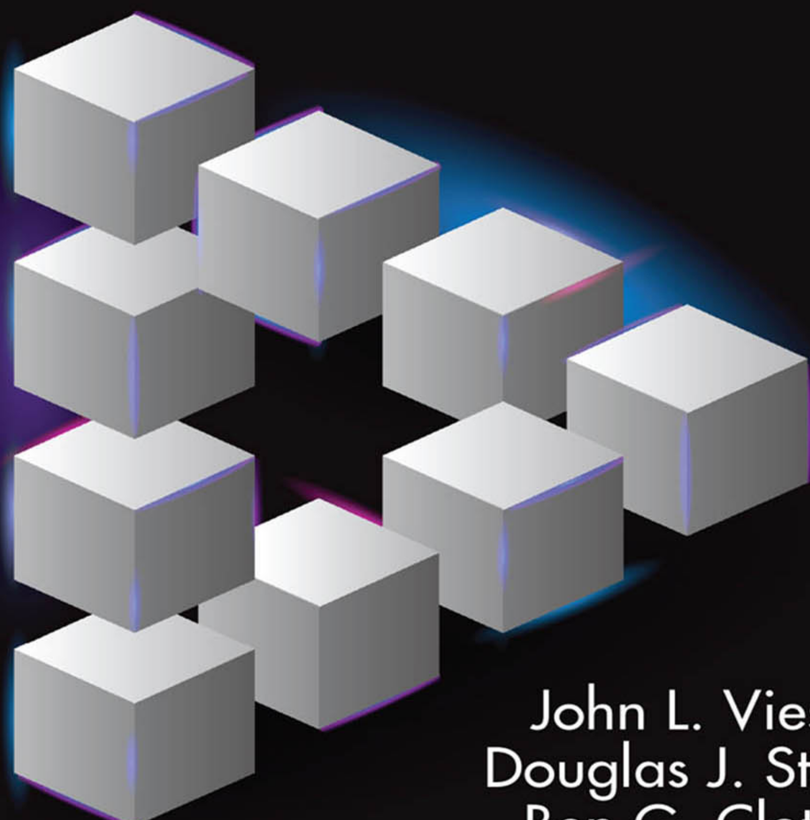
Effective Software Development Series

Scott Meyers, Consulting Editor

Effective SQL

61 Specific Ways to Write Better SQL



John L. Viescas
Douglas J. Steele
Ben G. Clothier

Foreword by Keith W. Hare

Praise for *Effective SQL*

"Given the reputation of the authors, I expected to be impressed. Impressed doesn't cover it, though. I was blown away! Most SQL books tell you 'how.' This one tells you 'why.' Most SQL books separate database design from implementation. This one integrates design considerations into every facet of SQL use. Most SQL books sit on my shelf. This one will live on my desk."

—Roger Carlson, Microsoft Access MVP (2006–2015)

"It can be easy to learn the basics of SQL, but it is very difficult to build accurate and efficient SQL, especially for critical systems with complex requirements. But now, with this great new book, you can get up to speed and write effective SQL much more quickly, no matter which DBMS you use."

—Craig S. Mullins, Mullins Consulting, Inc., DB2 Gold Consultant and IBM Champion for Analytics

"This is a great book. It is written in language that can be understood by a relative beginner and yet contains tips and tricks that will benefit the most hardened workhorse. It will therefore appeal to readers across the whole range of expertise and should be in the library of anybody who is seriously concerned with designing, managing, or programming databases."

—Graham Mandeno, database consultant and Microsoft MVP (1996–2015)

"This book is an excellent resource for database designers and developers working with relational and SQL-based databases—it's an easy read with great examples that combine theory with practical examples seamlessly. Examples for top relational databases Oracle, DB2, SQL Server, MySQL, and PostgreSQL are included throughout. The book walks the reader through sophisticated techniques to deal with things such as hierarchical data and tally tables, along with explanations of the inner workings and performance implications of SQL using `GROUP BY`, `EXISTS`, `IN`, correlated and non-correlated subqueries, window functions, and joins. The tips you won't find anywhere else, and the fun examples help to make this book stand out from the crowd."

—Tim Quinlan, database architect and Oracle Certified DBA

"This book is good for those who need to support multiple dialects of SQL. It's divided up into stand-alone items that you just grab and go. I have been doing SQL in various flavors since 1992 and even I picked up a few things."

—Tom Moreau, Ph.D., SQL Server MVP (2001–2012)

"This book is a powerful, compact, and easily understandable presentation of how to use SQL—it shows the application of SQL to real-world questions in order to teach the construction of queries, and it explains the relationship of 'how data is stored' to 'how data is queried' so that you obtain results successfully and effectively."

—Kenneth D. Snell, Ph.D., database consultant and former Microsoft Access MVP

"It has been problematic for many that there is no book on going from a novice database administrator to a much more advanced status until now. *Effective SQL* is a road map, a guide, a Rosetta Stone, and a coach on moving from basic Structured Query Language (SQL) to much more advanced uses to solve real-world problems. Rather than stumble around reinventing the wheel or catching glimpses of the proper ways to use a database, do yourself a favor and buy a copy of this book. Not only will you see many different approaches it would take years to see as a database consultant, but you will get a detailed understanding of why the databases of many vendors do what they do. Save time, effort, and wear and tear on your walls from banging your head against them and get this book."

—*Dave Stokes, MySQL Community Manager, Oracle Corporation*

"*Effective SQL* is a 'must have' for any serious database developer. It shows how powerful SQL can be in solving real-world problems in a step-by-step manner. The authors use easy-to-understand language in pointing out every advantage and disadvantage of each solution presented in the book. As we all know, there are multiple ways of accomplishing the same thing in SQL, but the authors explain why a particular query is more efficient than others. The part I liked best about the book is the summary at the end of each section, which reemphasizes the take-away points and reminds the reader which pitfalls to avoid. I highly recommend this book to all my fellow database developers."

—*Leo (theDBguy™), UtterAccess Moderator and Microsoft Access MVP*

"I think this is the book that is relevant not only for developers, but also for DBAs, as it talks about writing efficient SQL and various ways of achieving a desired result. In my opinion, this is a must-have book. Another reason to have this book is that it covers most of the commonly used RDBMSs, and so if someone is looking to transition from one RDBMS to another, this is the book to pick up. The authors have done a fantastic job. My heartiest congratulations to them."

—*Vivek Sharma, technologist, Hybrid Cloud Solutions, Core Technology and Cloud, Oracle Asia Pacific*

Effective SQL

Effective SQL

61 Specific Ways to Write Better SQL

John L. Viescas
Douglas J. Steele
Ben G. Clothier

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016955468

ISBN-13: 978-0-13-457889-7
ISBN-10: 0-13-457889-9

1 16

Editor-in-Chief
Greg Wiegand

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Technical Reviewers
Richard Anthony Broersma Jr.
Craig S. Mullins
Vivek Sharma
Dave Stokes
Morgan Tocker

Managing Editor
Sandra Schroeder

Full-Service Production Manager
Julie B. Nahil

Project Editor
Anna Popick

Copy Editor
Barbara Wood

Indexer
Richard Evans

Proofreader
Anna Popick

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

For Suzanne, forever and always . . .

—John Viescas

To my gorgeous and intelligent wife, Louise.
Thanks once again for putting up with me while
I wrote this (and all the other times, too!).

—Doug Steele

Couldn't have done it without support from you both,
Suzanne and Harold!

—Ben Clothier

This page intentionally left blank

Contents

Foreword

In the 30 years since the database language SQL was initially adopted as an international standard, the SQL language has been implemented in a multitude of database products. Today, SQL is everywhere. It is in high-performance transaction-processing systems, in smartphone applications, and behind Web interfaces. There is even a whole category of databases called NoSQL whose common feature is (or was) that they don't use SQL. As the NoSQL databases have added SQL interfaces, "No" is now interpreted as "Not Only" SQL.

Because of SQL's prevalence, you are likely to encounter SQL in multiple products and environments. One of the (perhaps valid) criticisms of SQL is that while it is similar across products, there are subtle differences. These differences result from different interpretations of the standard, different development styles, or different underlying architectures. To understand these differences, it is helpful to have examples that compare and contrast the subtle differences in SQL dialects. *Effective SQL* provides a Rosetta Stone for SQL queries, showing how queries can be written in different dialects and explaining the differences.

I often claim that the best way to learn something is by making mistakes. The corollary to this claim is that the people who know the most have made the most mistakes and have learned from others' mistakes. This book includes examples of incomplete and incorrect SQL queries with explanations of why they are incomplete and incorrect. This allows you to learn from mistakes others have made.

SQL is a powerful and complex database language. As a database consultant and a participant in both the U.S. and international SQL Standards committees, I've seen a lot of queries that did not take advantage of SQL's capabilities. Application developers who fully learn SQL's power and complexities can take full advantage of SQL's capabilities not only to build applications that perform well, but also to build those applications efficiently. The 61 specific examples in *Effective SQL* assist in this learning.

—Keith W. Hare
Senior Consultant, JCC Consulting, Inc.;
Vice Chair, INCITS DM32.2—the U.S. SQL Standards Committee;
Convenor, ISO/IEC JTC1 SC32 WG3—the International SQL Standards Committee

Acknowledgments

A famous politician once said that "it takes a village" to raise a child. If you've ever written a book—technical or otherwise—you know it takes a great team to turn your "child" into a successful book.

First, many thanks to our acquisitions editor and project manager, Trina MacDonald, who not only badgered John to follow up his successful *SQL Queries for Mere Mortals*® book with one for the Effective Software Development Series, but also shepherded the project through its many phases. John assembled a truly international team to help put the book together, and he personally thanks them for their diligent work. Special thanks to Tom Wickerath for his assistance both early in the project and later during technical review.

Trina handed us off to Songlin Qiu, our development editor, who ably helped us understand the ins and outs of writing an Effective Series book. Many thanks, Songlin, for your guidance.

Next, Trina rounded up a great set of technical editors who arduously went through and debugged our hundreds of examples and gave us great feedback. Thanks go to Morgan Tocker and Dave Stokes, MySQL; Richard Broersma Jr., PostgreSQL; Craig Mullins, IBM DB2; and Vivek Sharma, Oracle.

Along the way, series editor and author of the bestselling title *Effective C++, Third Edition,* Scott Meyers, stepped in and gave us invaluable advice about how to turn our items into truly effective advice. We hope we've made the father of the series proud.

Then the production team of Julie Nahil, Anna Popick, and Barbara Wood helped us whip the book into final shape for publication. We couldn't have done it without you!

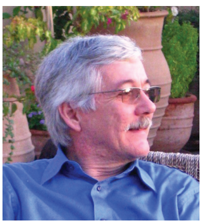
And finally, many thanks to our families who put up with many long nights while we worked on the manuscript and examples. Their enduring patience is greatly appreciated!

—John Viescas
Paris, France

—Doug Steele
St. Catharines, Ontario, Canada

—Ben Clothier
Converse, Texas, United States

About the Authors



John L. Viescas is an independent database consultant with more than 45 years of experience. He began his career as a systems analyst, designing large database applications for IBM mainframe systems. He spent six years at Applied Data Research in Dallas, Texas, where he directed a staff of more than 30 people and was responsible for research, product development, and customer support of database products for IBM mainframe computers. While working at Applied Data Research, John completed a degree in business finance at the University of Texas at Dallas, graduating cum laude.

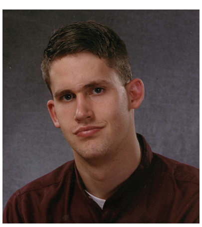
John joined Tandem Computers, Inc., in 1988, where he was responsible for the development and implementation of database marketing programs in Tandem's U.S. Western Sales region. He developed and delivered technical seminars on Tandem's relational database management system, NonStop SQL. John wrote his first book, *A Quick Reference Guide to SQL* (Microsoft Press, 1989), as a research project to document the similarities in the syntax among the ANSI-86 SQL Standard, IBM's DB2, Microsoft's SQL Server, Oracle Corporation's Oracle, and Tandem's NonStop SQL. He wrote the first edition of *Running Microsoft® Access* (Microsoft Press, 1992) while on sabbatical from Tandem. He has since written four editions of *Running*, three editions of *Microsoft® Office Access Inside Out* (Microsoft Press, 2003, 2007, and 2010)—the successor to the Running series, and *Building Microsoft® Access Applications* (Microsoft Press, 2005). He is also the best-selling author of *SQL Queries for Mere Mortals®, Third Edition* (Addison-Wesley, 2014). John currently holds the record for the most consecutive years being awarded MVP (Most Valuable Professional) for Microsoft Access from Microsoft, having received the award from 1993 to 2015. John makes his home with his wife of more than 30 years in Paris, France.

Douglas J. Steele has been working with computers, both mainframe and PC, for more than 45 years. (Yes, he did use punch cards in the beginning!) He worked for a large international oil company for more than 31 years before retiring in 2012. Databases and data modeling were a focus for most of that time, although he finished his career by developing the SCCM task sequence to roll Windows 7 out to over 100,000 computers worldwide.

Recognized by Microsoft as an MVP for more than 17 years, Doug has authored numerous articles on Access, was coauthor of *Microsoft® Access® Solutions: Tips, Tricks, and Secrets from Microsoft Access MVPs* (Wiley, 2010), and has been technical editor for a number of books.

Doug holds a master's degree in Systems Design Engineering from the University of Waterloo (Ontario, Canada), where his research centered on designing user interfaces for nontraditional computer users. (Of course, this was in the late seventies, so few people were traditional computer users at the time!) This research stemmed from his background in music (he holds an associateship in piano performance from the Royal Conservatory of Music, Toronto). He is also obsessed with beer and is a graduate of the Brewmaster and Brewery Operations Management program at Niagara College (Niagara-on-the-Lake, Ontario).

Doug lives with his lovely wife of more than 34 years in St. Catharines, Ontario. Doug can be reached at AccessMVPHelp@gmail.com.



Ben G. Clothier is a solution architect with IT Impact, Inc., a premier Access and SQL Server development shop based in Chicago, Illinois. He has worked as a freelance consultant with notable companies including J Street Technology and Advisicon and has worked on Access projects from small, one-person solutions to company-wide line-of-business applications. Notable projects include job tracking and inventory for a cement company, a Medicare insurance plan generator for an insurance provider, and order management for an international shipping company. Ben is an administrator at UtterAccess and was a coauthor, with Teresa Hennig, George Hepworth, and Doug Yudovich, of *Professional Access® 2013 Programming* (Wiley, 2013); a coauthor, with Tim Runcie and George Hepworth, of *Microsoft® Access in a SharePoint World* (Advisicon, 2011); and a contributing author of *Microsoft® Access® 2010 Programmer's Reference* (Wiley, 2010). He holds certifications for Microsoft SQL Server 2012 Solution Associate and MySQL 5.0 Certified Developer among others. He has been a Microsoft MVP since 2009.

Ben lives in San Antonio, Texas, with his wife, Suzanne, and his son, Harry.

About the Technical Editors

Richard Anthony Broersma Jr. is a systems engineer at Mangan, Inc., in Long Beach, California. He has 11 years of experience developing applications with PostgreSQL.

Craig S. Mullins is a data management strategist, researcher, and consultant. He is president and principal consultant of Mullins Consulting, Inc. Craig has been named by IBM as a Gold Consultant and an IBM Champion for Analytics. Craig has over three decades of experience in all facets of database systems development and has worked with DB2 since version 1. You may know Craig from his popular books: *DB2 Developer's Guide, Sixth Edition* (IBM Press, 2012) and *Database Administration: The Complete Guide to DBA Practices and Procedures, Second Edition* (Addison-Wesley, 2012).

Vivek Sharma is currently the designated "technologist" for the Oracle Core Technology and Hybrid Cloud Solutions Division at Oracle Asia Pacific. He has more than 15 years of experience working with Oracle technologies and started his career at Oracle as a developer working extensively on Oracle Forms and Reports before becoming a full-time Oracle DB performance architect. As an Oracle database expert, Vivek spends most of his time helping customers get the best out of their Oracle systems and database investments, and he is a member of the prestigious Oracle Elite Engineering Exchange and Server Technologies Partnership program. Sharma was declared "Speaker of the Year" in 2012 and 2015 by the Oracle India User Group Community. He writes articles on Oracle database technology on his blog, viveklsharma.wordpress.com, and for the Oracle Technology Network at www.oracle.com/technetwork/index.html.

Dave Stokes is a MySQL community manager for Oracle. Previously he was the MySQL certification manager for MySQL AB and Sun. He has worked for companies ranging alphabetically from the

American Heart Association to Xerox and done work ranging from anti-submarine warfare to Web developer.

Morgan Tocker is the product manager for MySQL Server at Oracle. He has previously worked in a variety of roles including support, training, and community. Morgan is based out of Toronto, Canada.

Introduction

Structured Query Language, or SQL, is the standard language for communicating with most database systems. We assume that because you are looking at this book, you have a need to get information from a database system that uses SQL.

This book is targeted at the application developers and junior database administrators (DBAs) who regularly work with SQL as part of their jobs. We assume that you are already familiar with the basic SQL syntax and focus on providing useful tips to get the most out of the SQL language. We have found that the mindset required is quite different from what works for computer programming as we move away from a procedural-based approach to solving problems toward a set-based approach.

A relational database management system (RDBMS) is a software application program you use to create, maintain, modify, and manipulate a relational database. Many RDBMS programs also provide the tools you need to create end-user applications that interact with the data stored in the database. RDBMS programs have continually evolved since their first appearance, and they are becoming more full-featured and powerful as advances occur in hardware technology and operating environments.

A Brief History of SQL

Dr. Edgar F. Codd (1923–2003), an IBM research scientist, first conceived the relational database model in 1969. He was looking into new ways to handle large amounts of data in the late 1960s and began thinking of how to apply mathematical principles to solve the myriad problems he had been encountering.

After Dr. Codd presented the relational database model to the world in 1970, organizations such as universities and research laboratories began efforts to develop a language that could be used as the foundation of a database system that supported the relational model. Initial work led to the development of several different languages in the early to mid-1970s. One such effort occurred at IBM's Santa Teresa Research Laboratory in San Jose, California.

IBM began a major research project in the early 1970s called System/R, intending to prove the viability of the relational model and to gain some experience in designing and implementing a relational database. Their initial endeavors between 1974 and 1975 proved successful, and they managed to produce a minimal prototype of a relational database.

At the same time they were working on developing a relational database, researchers were also working to define a database language. In 1974, Dr. Donald Chamberlin and his colleagues developed Structured English Query Language (SEQUEL), which allowed users to query a relational database using clearly defined English-style sentences. The initial success of their prototype database, SEQUEL-XRM, encouraged Dr. Chamberlin and his staff to continue their research. They revised SEQUEL into SEQUEL/2 between 1976 and 1977, but they had to change the name SEQUEL to SQL (Structured Query Language or SQL Query Language) for legal reasons—someone else had already used the acronym SEQUEL. To this day, many people still pronounce SQL as "sequel," although the widely accepted "official" pronunciation is "ess-cue-el."

Although IBM's System/R and SQL proved that relational databases were feasible, hardware technology at the time was not sufficiently powerful to make the product appealing to businesses.

In 1977 a group of engineers in Menlo Park, California, formed Relational Software, Inc., for the purpose of building a new relational database product based on SQL that they called Oracle. Relational Software shipped its product in 1979, providing the first commercially available RDBMS. One of Oracle's advantages was that it ran on Digital's VAX minicomputers instead of the more expensive IBM mainframes. Relational Software has since been renamed Oracle Corporation and is one of the leading vendors of RDBMS software.

At roughly the same time, Michael Stonebraker, Eugene Wong, and several other professors at the University of California's Berkeley computer laboratories were also researching relational database technology. They developed a prototype relational database that they named Ingres. Ingres included a database language called Query Language (QUEL), which was much more structured than SQL but made less use of English-like statements. However, it became clear that SQL was emerging as the standard database language, so Ingres was eventually converted to an SQL-based RDBMS. Several professors left Berkeley in 1980 to form Relational Technology, Inc., and in 1981 they announced the first commercial version of Ingres. Relational Technology has gone through several transformations. Formerly owned by Computer Associates International, Inc., and now part of Actian, Ingres is still one of the leading database products in the industry today.

Meanwhile, IBM announced its own RDBMS called SQL/Data System (SQL/DS) in 1981 and began shipping it in 1982. In 1983, the company introduced a new RDBMS product called Database 2 (DB2), which could be used on IBM mainframes using IBM's mainstream MVS operating system. First shipped in 1985, DB2 has become IBM's premier RDBMS, and its technology has been incorporated into the entire IBM product line.

With the flurry of activity surrounding the development of database languages, the idea of standardization was tossed about within the database community. However, no consensus or agreement as to who should set the standard or which dialect it should be based upon was ever reached, so each vendor continued to develop and improve its own database product in the hope that it—and, by extension, its dialect of SQL—would become the industry standard.

Customer feedback and demand drove many vendors to include certain elements in their SQL dialects, and in time an unofficial standard emerged. It was a small specification by today's standards, as it encompassed only those elements that were similar across the various SQL dialects. However, this specification (such as it was) did provide database customers with a core set of criteria by which to judge the various database programs on the market, and it also gave users knowledge that they could leverage from one database program to another.

In 1982, the American National Standards Institute (ANSI) responded to the growing need for an official relational database language standard by commissioning its X3 organization's database technical committee, X3H2, to develop a proposal for such a standard. After much effort (which included many improvements to SQL), the committee realized that its new standard had become incompatible with existing major SQL dialects, and the changes made to SQL did not improve it significantly enough to warrant the incompatibilities. As a result, they reverted to what was really just a minimal set of "least common denominator" requirements to which database vendors could conform.

ANSI ratified this standard, "ANSI X3.135-1986 Database Language SQL," which became commonly known as SQL/86, in 1986. In essence, it conferred official status on the elements that were similar among the various SQL dialects and that many database vendors had already implemented. Although the committee was aware of its shortcomings, at least the new standard provided a specific foundation from which the language and its implementations could be developed further.

The International Organization for Standardization (ISO) approved its own document (which corresponded exactly with ANSI SQL/86) as an international standard in 1987 and published it as “ISO 9075:1987 Database Language SQL.” (Both standards are still often referred to as just SQL/86.) The international database vendor community could now work from the same standards as vendors in the United States. Despite the fact that SQL gained the status of an official standard, the language was far from being complete.

SQL/86 was soon criticized in public reviews, by the government, and by industry pundits such as C. J. Date for problems such as redundancy within the SQL syntax (there were several ways to define the same query), lack of support for certain relational operators, and lack of referential integrity.

Both ISO and ANSI adopted refined versions of their standards in an attempt to address the criticisms, especially with respect to referential integrity. ISO published “ISO 9075: 1989 Database Language SQL with Integrity Enhancement” in mid-1989, and ANSI adopted its “X3.135-1989 Database Language SQL with Integrity Enhancement,” also often referred to as SQL/89, late that same year.

It was generally recognized that SQL/86 and SQL/89 lacked some of the most fundamental features needed for a successful database system. For example, neither standard specified how to make changes to the database structure once it was defined. It was not possible to modify or delete any structural component, or to make changes to the security of the database, despite the fact that all vendors provided ways to do this in their commercial products. (For example, you could `CREATE` a database object, but no `ALTER` or `DROP` syntax was defined.)

Not wanting to provide yet another “least common denominator” standard, both ANSI and ISO continued working on major revisions to SQL that would make it a complete and robust language. The new version (SQL/92) would include features that most major database vendors had already widely implemented, but it also included features that had not yet gained wide acceptance, as well as new features that were substantially beyond those currently implemented.

ANSI and ISO published their new SQL Standards—“X3.135-1992 Database Language SQL” and “ISO/IEC 9075:1992 Database Language

SQL," respectively—in October 1992. The SQL/92 document is considerably larger than the one for SQL/89, but it is also much broader in scope. For example, it provides the means to modify the database structure after it has been defined, supports additional operations for manipulating character strings as well as dates and times, and defines additional security features. SQL/92 was a major step forward from any of its predecessors.

While database vendors worked on implementing the features in SQL/92, they also developed and implemented features of their own, making additions to the SQL Standard known as "extensions." While the extensions (such as providing more data types than the six specified in SQL/92) provided more functionality within a given product and allowed vendors to differentiate themselves from one another, there were drawbacks. The main problem with adding extensions is that it causes each vendor's dialect of SQL to diverge further from the original standard, which prevents database developers from creating portable applications that can be run from any SQL database.

In 1997, ANSI's X3 organization was renamed the National Committee for Information Technology Standards (NCITS), and the technical committee in charge of the SQL Standard is now called ANSI NCITS-H2. Because of the rapidly growing complexity of the SQL Standard, the ANSI and ISO standards committees agreed to break the standard into 12 separate numbered parts and one addendum as they began to work on SQL3 (so named because it is the third major revision of the standard) so that work on each part could proceed in parallel. Since 1997, two additional parts have been defined.

Everything you read in this book is based on the current ISO Standard for the SQL database language—SQL/Foundation (document ISO/IEC 9075-2:2011)—as currently implemented in most of the popular commercial database systems. ANSI also adopted the ISO document, so this is truly an international standard. We also used the documentation from the latest versions of IBM DB2, Microsoft Access, Microsoft SQL Server, MySQL, Oracle, and PostgreSQL to provide, where necessary, syntax specific to each product. Although most of the SQL you will learn here is not specific to any particular software product, we do show you product-specific examples where appropriate.

Database Systems We Considered

Although you saw in the previous section that there are standards for SQL, that is not to say that all DBMSs are the same. The Web site DB-Engines collects and presents information on DBMSs and provides a monthly listing of them, ranked by their current popularity, at http://db-engines.com/en/ranking/relational+dbms.

For many months now, their rankings have presented six DBMSs as consistently the most popular, listed in alphabetical order here (the versions that we used for our testing are in parentheses):

1. IBM DB2 (DB2 for Linux, UNIX, and Windows v10.5.700.368)
2. Microsoft Access (Microsoft Access 2007—also compatible with versions 2010, 2013, 2016, and later)
3. Microsoft SQL Server (Microsoft SQL Server 2012—11.0.5343.0)
4. MySQL (MySQL Community Server 5.7.11)
5. Oracle Database (Oracle Database 11g Express Edition Release 11.2.0.2.0)
6. PostgreSQL (PostgreSQL 9.5.2)

That does not mean that the material presented in this book will not work on a DBMS not in that list of six. It simply means that we have not tested the material on other DBMSs or for different versions of the DBMSs listed. As you read this book, you will see that we have included advice (as Notes) when it is necessary to make changes. Those Notes apply only to the six DBMSs listed here. If you are using a different DBMS, check your documentation for compliance if you run into issues with any of our samples.

Sample Databases

To illustrate the concepts presented in this book, we use a number of sample databases, including the following:

1. **Beer Styles:** This is a fun attempt to catalog the details of 89 different styles of beer, based on the information presented by Michael Larson in his book *Beer: What to Drink Next* (Sterling Epicure, 2014).
2. **Entertainment Agency:** This database is designed to manage entertainers, agents, customers, and bookings. You would use a similar design to handle event bookings or hotel reservations.
3. **Recipes:** You can use this database to save and manage all your favorite recipes, as well as some of our favorites.
4. **Sales Orders:** This is a typical order-entry database for a store that sells bicycles, skateboards, and accessories.
5. **Student Grades:** This database lists students, the courses in which they are enrolled, and their performance in those courses.

We also provide a number of sample databases specific to a particular item, some of which are built by a code listing within the item. The schemas and sample data are available in the GitHub site associated with the book.

Where to Find the Samples on GitHub

Many technical books come with a CD-ROM containing the examples in electronic form. That can be limiting, so we decided to provide our examples in GitHub, at https://github.com/TexanInParis/Effective-SQL.

There, you will find high-level folders for each of the six DBMSs we considered. Within each of those high-level folders are ten folders corresponding to the ten chapters in the book, plus a folder for the sample databases.

Within each of the ten chapter folders, there are individual files, named to correspond to the listing numbers within each chapter. Note that not all listings are applicable to every DBMS. When that is the case, we highlight differences in the README files for each chapter. For Microsoft Access, the README file indicates which sample database contains the listings for the chapter.

The root folder on GitHub also contains the Listings.xlsx file that shows you which database contains each listing. That file also documents SQL samples that are applicable to each of the six database systems.

Each of the sample database folders, with the exception of the Microsoft Access folder that contains .accdb files in 2007 format, contains a number of SQL files. We used the 2007 format for Microsoft Access because it is compatible over all versions of the product since version 12 (2007). One set of these files creates the structure for each sample database, and the other set of files contains the data to populate the sample databases. (Note that some of the items in this book rely on specific data cases. The structures and data for those items are sometimes contained within the chapter listings.)

Note

In preparing the listings in this book for publication, we sometimes had issues fitting within the 63-character-per-line limit imposed by the physical page. It is possible that a listing could have been edited incorrectly. When in doubt, all the listings on GitHub were tested, so we are confident that they are correct.

Summary of the Chapters

As the title of the book suggests, 61 specific items are presented in this book. Each item is intended to stand by itself; you should not need to read other items in order to use the material presented in a specific item. There are, of course, times when the material in a specific item does build on material in other items. When that is the case, we have tried to present as much background material as we felt was necessary, but we do provide cross-references to other relevant items so that you can review the material yourself.

Although each item is, as already stated, intended to stand alone, we felt there were natural groupings of topics. The groupings we used are these ten:

1. **Data Model Design:** Because you cannot write effective SQL when you are working with a bad data model design, the items in this chapter cover some basics of good relational model design. If your database design violates any of the rules discussed in this chapter, you need to figure out what is wrong and fix it.
2. **Programmability and Index Design:** Simply having a good logical data model design is not sufficient to allow you to write effective SQL. You must ensure that you have implemented the design in an appropriate manner, or you may find that your ability to extract meaningful information from the data in an efficient manner using SQL will be compromised. The items in this chapter help you understand the importance of indexes, and how to ensure that they have been properly implemented.
3. **When You Can't Change the Design:** Sometimes, despite your best efforts, you are forced to deal with external data outside of your control. The items in this chapter are intended to help you deal with such situations.
4. **Filtering and Finding Data:** The ability to look for or filter out the data of interest is one of the most important tasks you can do in SQL. The items in this chapter explore different techniques you can use to extract the exact information you want.
5. **Aggregation:** The SQL Standard has always provided the ability to aggregate data. However, typically you are asked to provide "totals per customer," "count of orders by day," or "average sales of each category by month." It is the part after the "per," "by," and "of each" that requires additional attention. The items in this chapter present techniques to get the best performance out of your aggregation. Some of them also show how to use window functions to provide even more complex aggregations.

6. **Subqueries:** There are many different ways in which you can use subqueries. The items in this chapter are intended to show a variety of ways to get additional flexibility in your SQL through the use of subqueries.
7. **Getting and Analyzing Metadata:** Sometimes just data is not enough. You need data about data. You might even need data about how you are getting the data. In some cases, it might even be convenient to get the metadata using SQL. The items in this chapter tend to be quite product specific, but our hope is that we provide sufficient information so that you can apply the principles to your specific DBMS.
8. **Cartesian Products:** Cartesian Products are the result of combining all rows in one table with all rows in a second table. While perhaps not as common as other join types, the items in this chapter show real-world situations where it would not be possible to answer the underlying question without the use of a Cartesian Product.
9. **Tally Tables:** Another useful tool is the tally table, usually a table with a single column of sequential numbers, or a single column of sequential dates, or something more complex to aid in "pivoting" a set of summaries. While Cartesian Products are dependent on actual values in the underlying tables, tally tables allow you to cover all possibilities. The items in this chapter show examples of various problems that can be solved only through the use of a tally table.
10. **Modeling Hierarchical Data:** It is not uncommon to have to model hierarchical data in your relational database. Unfortunately, it happens to be one of SQL's weaker areas. The items in this chapter are intended to help you make the trade-off between data normalization, and ease of querying and maintenance of metadata.

Each database system has a variety of functions that you can use to calculate or manipulate date and time values. Each database system also has its own rules regarding data types and date and time arithmetic. Because of the differences, we also included an Appendix, "Date and Time Types, Operations, and Functions," to help you work with date and time values in your database system. We believe it accurately summarizes the data types and arithmetic operations supported, but we do recommend that you consult your database documentation for the specific syntax to use with each function.

This page intentionally left blank

3 When You Can't Change the Design

You have spent considerable time ensuring that you have a proper logical data model for your situation. You have worked hard to ensure that it has been implemented as an appropriate physical model. Unfortunately, you find that some of your data must come from a source outside your control.

This does not mean that you are doomed to have SQL queries that will not perform well. The items in this chapter are intended to help you understand some options you have to be able to work with that inappropriately designed data from other sources. We will consider both the case when you can create objects to hold the transformations and the case when you must perform the transformation as part of the query itself.

Because you do not have control over the external data, there is nothing you can do to change the design. However, you can use the information in the items in this chapter to work with the DBAs and still end up with effective SQL.

Item 18: Use Views to Simplify What Cannot Be Changed

Views are simply a composition of a table in the form of a predefined SQL query on one or many tables or other views. Although they are simple, there is much merit to their use.

> **Note**
>
> Microsoft Access does not actually have an object called a view, but saved queries in Access can be thought of as views.

You can use views to ameliorate some denormalization issues. You have already seen the denormalized `CustomerSales` table in Item 2, "Eliminate redundant storage of data items," and how it should have been modeled as four separate tables (`Customers`, `AutomobileModels`, `SalesTransactions`, and `Employees`). You've also seen the `Assignments` table with repeating groups in Item 3, "Get rid of repeating groups," that should have been modeled as two separate tables (`Drawings` and `Predecessors`). While working to fix such problems, you could use views to represent how the data should appear.

You can create different views of `CustomerSales` as shown in Listing 3.1.

Listing 3.1 Views to normalize a denormalized table

```
CREATE VIEW vCustomers AS
SELECT DISTINCT cs.CustFirstName, cs.CustLastName, cs.Address,
  cs.City, cs.Phone
FROM CustomerSales AS cs;

CREATE VIEW vAutomobileModels AS
SELECT DISTINCT cs.ModelYear, cs.Model
FROM CustomerSales AS cs;

CREATE VIEW vEmployees AS
SELECT DISTINCT cs.SalesPerson
FROM CustomerSales AS cs;
```

As Figure 3.1 shows, `vCustomers` would still include two entries for Tom Frank because two different addresses were listed in the original table. However, you have a smaller set of data to work with. By sorting the data on `CustFirstName` and `CustLastName`, you should be able to see the duplicate entry, and you can correct the data in the `CustomerSales` table.

vCustomers

CustFirstName	CustLastName	Address	City	Phone
Amy	Bacock	111 Dover Lane	Chicago	312-222-1111
Barney	Killjoy	4655 Rainier Ave.	Auburn	253-111-2222
Debra	Smith	3223 SE 12th Pl.	Seattle	206-333-4444
Homer	Tyler	1287 Grady Way	Renton	425-777-8888
Tom	Frank	7435 NE 20th St.	Bellevue	425-888-9999
Tom	Frank	7453 NE 20th St.	Bellevue	425-888-9999

Figure 3.1 Data for view `vCustomers`

You saw in Item 3 how to use a `UNION` query to "normalize" a table that contains repeating groups. You can use views to do the same thing, as shown in Listing 3.2.

Listing 3.2 Views to normalize a table with repeating groups

```
CREATE VIEW vDrawings AS
SELECT a.ID AS DrawingID, a.DrawingNumber
FROM Assignments AS a;

CREATE VIEW vPredecessors AS
SELECT 1 AS PredecessorID, a.ID AS DrawingID,
  a.Predecessor_1 AS Predecessor
FROM Assignments AS a
WHERE a.Predecessor_1 IS NOT NULL
UNION
SELECT 2, a.ID, a.Predecessor_2
FROM Assignments AS a
WHERE a.Predecessor_2 IS NOT NULL
UNION
SELECT 3, a.ID, a.Predecessor_3
FROM Assignments AS a
WHERE a.Predecessor_3 IS NOT NULL
UNION
SELECT 4, a.ID, a.Predecessor_4
FROM Assignments AS a
WHERE a.Predecessor_4 IS NOT NULL
UNION
SELECT 5, a.ID, a.Predecessor_5
FROM Assignments AS a
WHERE a.Predecessor_5 IS NOT NULL;
```

One point that needs to be mentioned is that although all the views shown previously mimic what the proper table design should be, they can be used only for reporting purposes. Because of the use of SELECT DISTINCT in the views in Listing 3.1, and the use of UNION in Listing 3.2, the views are not updatable. Some vendors allow you to work around this limitation by defining triggers on views (also known as INSTEAD OF triggers) so that you can write the logic for applying modifications made via the view to the underlying base table yourself.

> **Note**
>
> DB2, Oracle, PostgreSQL, and SQL Server allow triggers on views. MySQL does not.

Some other reasons to use views include the following:

- **To focus on specific data**: You can use views to focus on specific data and on specific tasks. The view can return all rows of a table or tables, or a WHERE clause can be included to limit the rows returned. The view can also return only a subset of the columns in one or more tables.

- **To simplify or clarify column names**: You can use views to provide aliases on column names so that they are more meaningful.
- **To bring data together from different tables**: You can use views to combine multiple tables into a single logical record.
- **To simplify data manipulation**: Views can simplify how users work with data. For example, assume you have a complex query that is used for reporting purposes. Rather than make each user define the subqueries, outer joins, and aggregation to retrieve data from a group of tables, create a view. Not only does the view simplify access to the data (because the underlying query does not have to be written each time a report is being produced), but it ensures consistency by not forcing each user to create the query. You can also create inline user-defined functions that logically operate as parameterized views, or views that have parameters in `WHERE` clause search conditions or other parts of the query. Note that inline table-valued functions are not the same as scalar functions!
- **To protect sensitive data**: When the table contains sensitive data, that data can be left out of the view. For instance, rather than reveal customer credit card information, you can create a view that uses a function to "munge" the credit card numbers so that users are not aware of the actual numbers. Depending on the DBMS, only the view would be made accessible to users, and the underlying tables need not be directly accessible. Views can be used to provide both column-level and row-level security. Note that a `WITH CHECK OPTION` clause is necessary to protect the data integrity by preventing users from performing updates or deletes that go beyond the constraints imposed by the view.
- **To provide backward compatibility**: Should changes be required to the schemas for one or more of the tables, you can create views that are the same as the old table schemas. Applications that used to query the old tables can now use the views, so that the application does not have to be changed, especially if it is only reading data. Even applications that update data can sometimes still use a view if `INSTEAD OF` triggers are added to the new view to map `INSERT`, `DELETE`, and `UPDATE` operations on the view to the underlying tables.
- **To customize data**: You can create views so that different users can see the same data in different ways, even when they are using the same data at the same time. For example, you can create a

view that retrieves only the data for those customers of interest to a specific user based on that user's login ID.

- **To provide summarizations**: Views can use aggregate functions (SUM(), AVERAGE(), etc.) and present the calculated results as part of the data.
- **To export and import data**: You can use views to export data to other applications. You can create a view that gives you only the desired data, and then use an appropriate data utility to export just that data. You can also use views for import purposes when the source data does not contain all columns in the underlying table.

Do Not Create Views on Views

It is permissible to create a view that references another view(s). Those coming from a programming background might be tempted to treat a view the way they would treat a procedure in an imperative programming language. That is actually a big mistake and will cause more performance and maintenance problems, likely offsetting any savings gained from having a generic view that is then used as a base for other views. Listing 3.3 demonstrates an example of creating views on other views.

Listing 3.3 Three view definitions

```
CREATE VIEW vActiveCustomers AS
SELECT c.CustomerID, c.CustFirstName, c.CustLastName,
  c.CustFirstName + ' ' + c.CustLastName AS CustFullName
FROM Customers AS c
WHERE EXISTS
  (SELECT NULL
   FROM Orders AS o
   WHERE o.CustomerID = c.CustomerID
     AND o.OrderDate > DATEADD(MONTH, -6, GETDATE()));

CREATE VIEW vCustomerStatistics AS
SELECT o.CustomerID, COUNT(o.OrderNumber) AS OrderCount,
  SUM(o.OrderTotal) AS GrandOrderTotal,
  MAX(o.OrderDate) AS LastOrderDate
FROM Orders AS o
GROUP BY o.CustomerID;
```

continues

```
CREATE VIEW vActiveCustomerStatistics AS
SELECT a.CustomerID, a.CustFirstName, a.CustLastName,
  s.LastOrderDate, s.GrandOrderTotal
FROM vActiveCustomers AS a
  INNER JOIN vCustomerStatistics AS s
    ON a.CustomerID = s.CustomerID;
```

There are several potential issues, not all of which might be manifested the same way on different vendors' products. However, generally speaking, giving the optimizer the view as the source means that the optimizer has to first decompose the view. If there are other view references, those must also be decomposed. In an ideal implementation, the optimizer would efficiently "inline" the three view definitions into the equivalent statement in Listing 3.4.

Listing 3.4 Equivalent statement of combined views

```
SELECT c.CustomerID, c.CustFirstName, c.CustLastName,
  s.LastOrderDate, s.GrandOrderTotal
FROM Customers AS c
  INNER JOIN
    (SELECT o.CustomerID,
        SUM(o.OrderTotal) AS GrandOrderTotal,
        MAX(o.OrderDate) AS LastOrderDate
     FROM Orders AS o
     GROUP BY o.CustomerID) AS s
    ON c.CustomerID = s.CustomerID
WHERE EXISTS
  (SELECT NULL
   FROM Orders AS o
   WHERE o.CustomerID = c.CustomerID
     AND o.OrderDate > DATEADD(MONTH, -6, GETDATE()));
```

Note that certain columns or expressions are already pruned from Listing 3.4 where they are actually not used. In particular, `OrderCount` and `CustFullName` were not present anywhere within the main query and subquery. However, in practice the optimizer might be forced to preprocess the views completely, including evaluating all expressions in order to create intermediate results for joining to other intermediate results. Because the final view did not use them all, some expressions were discarded in spite of all the hard work put into calculating them.

The same concerns apply to the rows that could be filtered. For example, inactive customers were included in vCustomerStatistics but ultimately were not in the final view because vActiveCustomers excluded those customers. This can potentially result in far more I/Os than you anticipated. You can learn more about those considerations in Item 46, "Understand how the execution plan works." Although this is a somewhat oversimplified example, it is fairly easy to create a view that the optimizer simply cannot inline when it is referenced in other views. Worse, there would be more than one way to create such views that would prevent inlining. Finally, the optimizer generally does a better job when it is given a simpler query expression that asks for exactly the data it actually needs.

For those reasons, it is best to avoid creating views on views. If you need a different presentation of the view, create a new view that directly references the base tables with the appropriate filters or groupings applied. You can also embed subqueries in a view, which can be useful in making the aggregated calculations "private" to the view. This approach helps to prevent proliferation of several views that are not directly usable, making the database solution much more maintainable. Refer to Item 42, "If possible, use common table expressions instead of subqueries," for additional techniques.

Things to Remember

- Use views to structure data in a way that users will find natural or intuitive.
- Use views to restrict access to the data such that users can see (and sometimes modify) exactly what they need and no more. Remember to use `WITH CHECK OPTION` when necessary.
- Use views to hide and reuse complex queries.
- Use views to summarize data from various tables that can be used to generate reports.
- Use views to implement and enforce naming and coding standards, especially when working with legacy database designs that need to be updated.

Item 19: Use ETL to Turn Nonrelational Data into Information

Extract, Transform, Load (ETL) is a set of procedures or tools you can use to **E**xtract data from an external source, **T**ransform it to conform to relational design rules or to conform to other requirements, then **L**oad it into your database for further use or analysis. Nearly all database systems provide various utilities to aid in this process. These utilities are, quite simply, a means to convert raw data into information.

To get an idea of what these utilities can do, let's take a look at some of the tools in Microsoft Access—one of the first Windows-era database systems to provide built-in ways to load and transform data into something useful. Assume you work as the marketing manager for a company that produces breakfast cereals. You need not only to analyze competitive sales from another manufacturer but also to break down this analysis by individual brands.

You can certainly glean total sales information from publicly available documents, but you really want to try to break down competitive sales by individual brand. To do this, you might strike up an agreement with a major grocery store chain to get them to provide their sales information by brand in return for a small discount on your products. The grocery chain promises to send you a spreadsheet containing sales data from all its stores broken down by competitive brand for the previous year. The data you receive might look like Table 3.1.

Table 3.1 Sample competitive sales data

Product		**Jan**		**Feb**		**Mar**
Alpha-Bits		57775.87		40649.37		. . .
Golden Crisp		33985.68		17469.97		. . .
Good Morenings		40375.07		36010.81		. . .
Grape-Nuts		55859.51		38189.64		. . .
Great Grains		37198.23		41444.41		. . .
Honey Bunches of Oats		63283.28		35261.46		. . .
. . . additional rows . . .						

It is clear that some blank columns that you do not need were added for readability. You also need to transform the data to end up with one row per product per month, and you have a separate table listing competitive products that has its own primary key, so you need to match on product name to get the key value to use as a foreign key.

Let's start by extracting the data from the spreadsheet into a more usable form. Microsoft Access can import data in many different formats, so let's fire up the Import tool to import a spreadsheet. In the first step, you identify the file and tell Access what you want to do with the output (import into a new table, append the data to an existing table, or link as a read-only table).

When you go to the next step, Access shows you a grid with a sample of the data it found, as shown in Figure 3.2. Because it determined that the first row might very well be usable as column names, it has used the names it found and has assigned generated names to the blank columns.

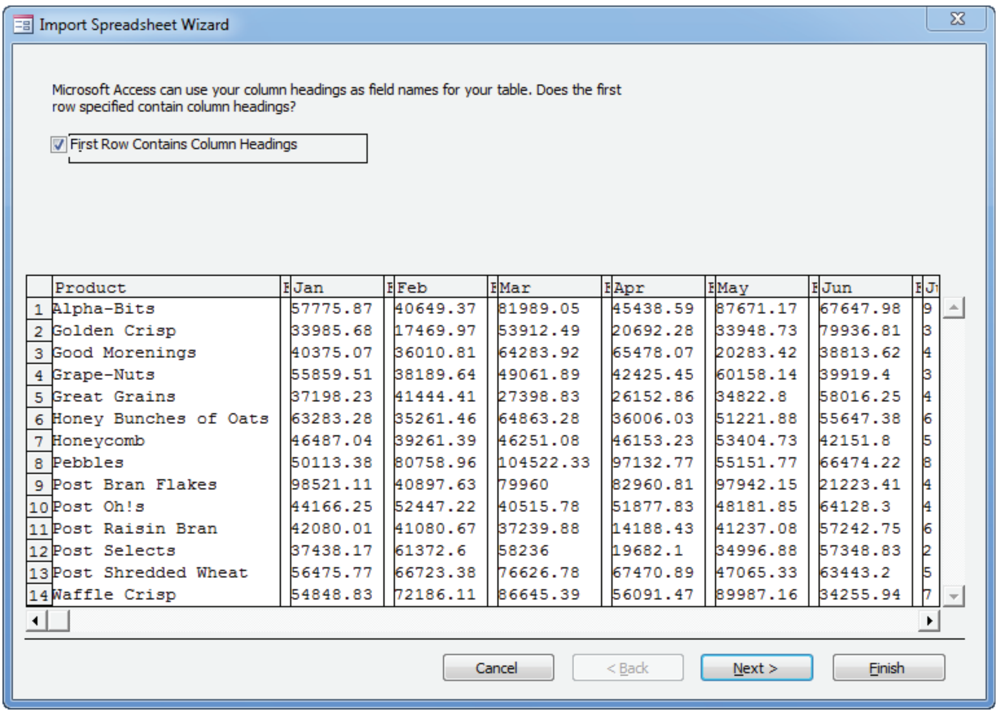


Figure 3.2 The Import Spreadsheet utility performing an initial analysis of the data

In the following step, Access shows you a display where you can select columns one at a time, tell Access to skip unimportant columns, and fix the data type that the utility has assumed. Figure 3.3 on the next page shows one of the data columns selected. The utility has assumed that the numbers, because they contain decimal points, should be imported as the very flexible `Double` data type. We know that these are all dollar sales figures, so it makes sense to change the data type to `Currency` to make it easier to work with the data. You can also see the "Do not import" check box (behind the drop-down) that you can select for columns that you want to ignore.

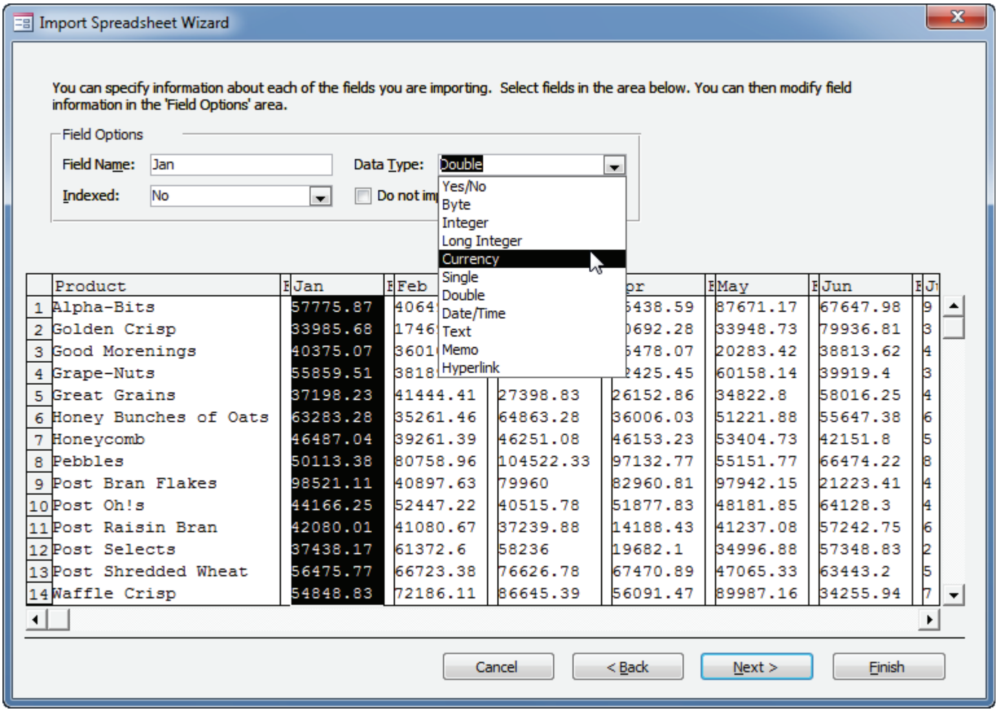


Figure 3.3 Selecting columns to skip and choosing a data type

The next step in the utility lets you pick a column to act as the primary key, ask the utility to generate an ID column with incrementing integers, or assign no primary key to the table. The final step allows you to name the table (the default is the name of the worksheet) and to invoke another utility after importing the table to perform further analysis and potentially reload the data into a more normalized table design. If you choose to run the Table Analyzer, Access presents you with a design tableau as shown in Figure 3.4. In the figure, we have already dragged and dropped the Product column into a separate table and named both tables. As you can see, the utility automatically generates a primary key in the product table and provides a matching foreign key in the sales data table.

Even after using the Table Analyzer, you can see that there is still plenty of work to do to further normalize the sales data into one row per month. You can "unpivot" the sales data by using a `UNION` query to turn the columns into rows, as shown in Listing 3.5. (See also Item 21, "Use `UNION` statements to 'unpivot' non-normalized data.")

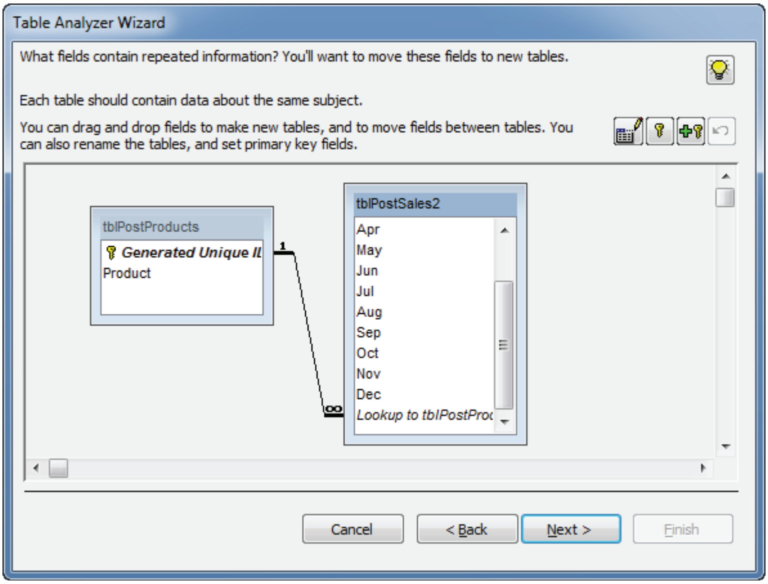


Figure 3.4 Using the Table Analyzer to break out products into a separate table

Listing 3.5 Using a UNION query to "unpivot" a repeating group

```
SELECT '2015-01-01' AS SalesMonth, Product, Jan AS SalesAmt
FROM tblPostSales
UNION ALL
SELECT '2015-02-01' AS SalesMonth, Product, Feb AS SalesAmt
FROM tblPostSales
UNION ALL
  ... etc. for all 12 months.
```

The tools in Microsoft Access are fairly simple (for example, they cannot handle totals rows), but they give you an idea of the amount of work that can be saved when trying to perform ETL to load external data into your database. As noted earlier, most database systems provide similar—and in some cases more powerful—tools that you can use. Examples include Microsoft SQL Server Integration Services (SSIS), Oracle Data Integrator (ODI), and IBM InfoSphere DataStage. Commercial tools are available from vendors such as Informatica, SAP, and SAS, and you can also find a number of open-source tools available on the Web.

The main point here is that you should use those tools so that your data conforms to the data model that your business needs, not the other way around. A common mistake is to build tables that fit the incoming data as is and then use it directly in applications. The investment made to transform data will result in a database that is easy to understand and maintain in spite of the divergent data sources from which it may collect the raw input.

Things to Remember

- ETL tools allow you to import nonrelational data into your database with less effort.
- ETL tools help you reformat and rearrange imported data so that you can turn it into information.
- Most database systems offer some level of ETL tools, and there are also commercial tools available.

Item 20: Create Summary Tables and Maintain Them

We mentioned previously (in Item 18, "Use views to simplify what cannot be changed") that views can be used to simplify complex queries, and we even suggested that views can be used to provide summarizations. Depending on the volume of data, there are times when it may be more appropriate to create summary tables.

When you have a summary table, you can be sure that everything is in one place, making it easier to understand the data structure and quicker to return information.

One approach is to create a table that summarizes your data in your details table, and write triggers to update the summary table every time something changes in the details table. However, if your details table is frequently modified, this can be processor intensive.

Another approach is to use a stored procedure to refresh the summary table on a regular basis: delete all existing data rows and reinsert the summarized information.

DB2 has the concept of summary tables built into it. DB2 summary tables can maintain a summary of data in one or multiple tables. You have the option to have the summary refreshed every time the data in underlying table(s) changes, or you can refresh it manually. DB2 summary tables not only allow users to obtain results faster, but the optimizer can use the summary tables when user queries indirectly request information already summarized in the summary tables if `ENABLE QUERY OPTIMIZATION` is specified when you create the summary table. Although there may still be "costs" associated with all that activity, at least you did not have to write triggers or stored procedures to maintain the data for you.

Listing 3.6 shows how to create a summary table named SalesSummary that summarizes data from six different tables in DB2. Note that the SQL is not much different from that for creating a view. In fact, a summary table is a specific type of materialized query table, identified by the inclusion of a GROUP BY clause in the CREATE SQL. Note that we had to use Cartesian joins with filters, because of the restriction against using INNER JOIN in a materialized query table, and additionally provide COUNT(*) in the SELECT list to enable the use of the REFRESH IMMEDIATE clause. Those are necessary to permit the optimizer to use it.

Listing 3.6 Creating a summary table based on six tables (DB2)

```
CREATE SUMMARY TABLE SalesSummary AS (
SELECT
  t5.RegionName AS RegionName,
  t5.CountryCode AS CountryCode,
  t6.ProductTypeCode AS ProductTypeCode,
  t4.CurrentYear AS CurrentYear,
  t4.CurrentQuarter AS CurrentQuarter,
  t4.CurrentMonth AS CurrentMonth,
  COUNT(*) AS RowCount,
  SUM(t1.Sales) AS Sales,
  SUM(t1.Cost * t1.Quantity) AS Cost,
  SUM(t1.Quantity) AS Quantity,
  SUM(t1.GrossProfit) AS GrossProfit
FROM Sales AS t1, Retailer AS t2, Product AS t3,
  datTime AS t4, Region AS t5, ProductType AS t6
WHERE t1.RetailerId = t2.RetailerId
  AND t1.ProductId = t3.ProductId
  AND t1.OrderDay = t4.DayKey
  AND t2.RetailerCountryCode = t5.CountryCode
  AND t3.ProductTypeId = t6.ProductTypeId
GROUP BY t5.RegionName, t5.CountryCode, t6.ProductTypeCode,
  t4.CurrentYear, t4.CurrentQuarter, t4.CurrentMonth
)
DATA INITIALLY DEFERRED
REFRESH IMMEDIATE
ENABLE QUERY OPTIMIZATION
MAINTAINED BY SYSTEM
NOT LOGGED INITIALLY;
```

Listing 3.7 on the next page shows how to provide a similar capability in Oracle through the use of a materialized view.

Listing 3.7 Creating a materialized view based on six tables (Oracle)

```
CREATE MATERIALIZED VIEW SalesSummary
  TABLESPACE TABLESPACE1
  BUILD IMMEDIATE
  REFRESH FAST ON DEMAND
AS
SELECT SUM(t1.Sales) AS Sales,
  SUM(t1.Cost * t1.Quantity) AS Cost,
  SUM(t1.Quantity) AS Quantity,
  SUM(t1.GrossProfit) AS GrossProfit,
  t5.RegionName AS RegionName,
  t5.CountryCode AS CountryCode,
  t6.ProductTypeCode AS ProductTypeCode,
  t4.CurrentYear AS CurrentYear,
  t4.CurrentQuarter AS CurrentQuarter,
  t4.CurrentMonth AS CurrentMonth
FROM Sales AS t1
  INNER JOIN Retailer AS t2
    ON t1.RetailerId = t2.RetailerId
  INNER JOIN Product AS t3
    ON t1.ProductId = t3.ProductId
  INNER JOIN datTime AS t4
    ON t1.OrderDay = t4.DayKey
  INNER JOIN Region AS t5
    ON t2.RetailerCountryCode = t5.CountryCode
  INNER JOIN ProductType AS t6
    ON t3.ProductTypeId = t6.ProductTypeId
GROUP BY t5.RegionName, t5.CountryCode, t6.ProductTypeCode,
  t4.CurrentYear, t4.CurrentQuarter, t4.CurrentMonth;
```

Although SQL Server does not directly support materialized views, the fact that you can create indexes on views has a similar effect, and thus you can use indexed views in a similar manner.

Note

Various vendors implement additional restrictions. We advise first consulting your documentation to determine what is actually supported before creating a summary table/materialized view/indexed view.

Note that there can be some negative aspects to summary tables as well, such as the following:

- Each summary table occupies storage.
- The administrative work (triggers, constraints, stored procedures) may need to exist on both the original table and any summary tables.

- You need to know in advance what users want to query in order to precompute the required aggregations and include them in the summary tables.
- You may need multiple summary tables if you need different groupings or filters applied.
- You may need to set up a schedule to manage the refresh of the summary tables.
- You may need to manage the periodicity of the summary tables via SQL. For example, if the summary table is supposed to show the past 12 months, you need a way to remove data that is more than a year old from the table.

One possible suggestion to avoid some of the increased administrative costs of having redundant triggers, constraints, and stored procedures is to use what Ken Henderson referred to as inline summarization in his book *The Guru's Guide to Transact-SQL* (Addison-Wesley, 2000). This involves adding aggregation columns to the existing table. You would use an `INSERT INTO` SQL statement to aggregate data and store those aggregations in the same table. Columns that are not part of the aggregated data would be set to a known value (such as `NULL` or some fixed date). An advantage of doing inline summarization is that the summary and the detail data can be easily queried together or separately. The summarized records are easily identified by the known values in certain columns, but other than that, they are indistinguishable from the detail records. However, this approach necessitates that all queries on the table containing both detail and summary data be written appropriately.

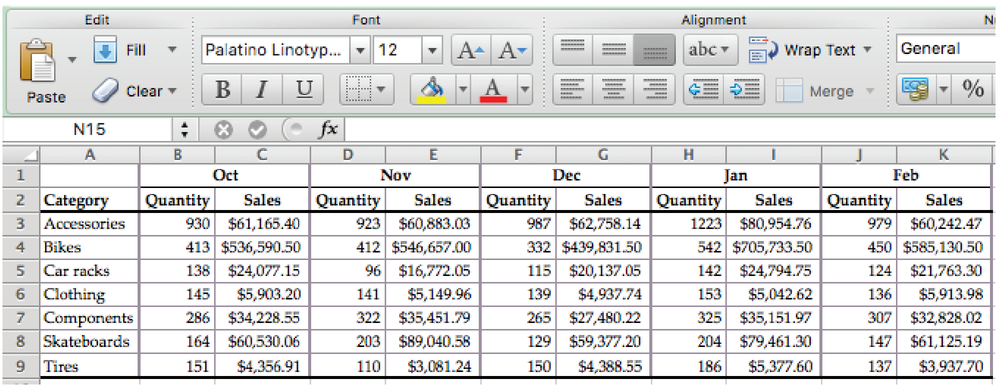
Things to Remember

- ✦ Storing summarized data can help minimize the processing required for aggregation.
- ✦ Using tables to store the summarized data allows you to index fields containing the aggregated data for more efficient queries on aggregates.
- ✦ Summarization works best on tables that are more or less static. If the source tables change too often, the overhead of summarization may be too great.
- ✦ Triggers can be used to perform summarization, but a stored procedure to rebuild the summary table is usually better.

Item 21: Use UNION Statements to “Unpivot” Non-normalized Data

You saw in Item 3, “Get rid of repeating groups,” how UNION queries can be used to deal with repeating groups. We explore UNION queries a little bit more in this item. As you will learn in Item 22, “Understand relational algebra and how it is implemented in SQL,” the Union operation is one of the eight relational algebra operations that can be performed within the relational model defined by Dr. Edgar F. Codd. It is used to merge data sets created by two (or more) SELECT statements.

Assume that the only way you are able to get some data for analysis is in the form of the Excel spreadsheet pictured in Figure 3.5, which is obviously not normalized.



	A	B	C	D	E	F	G	H	I	J	K
1		Oct		Nov		Dec		Jan		Feb	
2	Category	Quantity	Sales	Quantity	Sales	Quantity	Sales	Quantity	Sales	Quantity	Sales
3	Accessories	930	$61,165.40	923	$60,883.03	987	$62,758.14	1223	$80,954.76	979	$60,242.47
4	Bikes	413	$536,590.50	412	$546,657.00	332	$439,831.50	542	$705,733.50	450	$585,130.50
5	Car racks	138	$24,077.15	96	$16,772.05	115	$20,137.05	142	$24,794.75	124	$21,763.30
6	Clothing	145	$5,903.20	141	$5,149.96	139	$4,937.74	153	$5,042.62	136	$5,913.98
7	Components	286	$34,228.55	322	$35,451.79	265	$27,480.22	325	$35,151.97	307	$32,828.02
8	Skateboards	164	$60,530.06	203	$89,040.58	129	$59,377.20	204	$79,461.30	147	$61,125.19
9	Tires	151	$4,356.91	110	$3,081.24	150	$4,388.55	186	$5,377.60	137	$3,937.70

Figure 3.5 Non-normalized data from Excel

Assuming you can import that data into your DBMS, at best you will end up with a table (SalesSummary) that has five pairs of repeating groups, which we will call OctQuantity, OctSales, NovQuantity, NovSales, and so on to FebQuantity and FebSales.

Listing 3.8 shows a query that would let you look at the October data.

Listing 3.8 SQL to extract October data

```
SELECT Category, OctQuantity, OctSales
FROM SalesSummary;
```

Of course, to look at the data for a different month, you need a different query. And let’s not forget that data that is not normalized can be more difficult to use for analysis purposes. This is where a UNION query can help.

There are three basic rules that apply when using UNION queries:

1. There must be the same number of columns in each of the queries making up the UNION query.
2. The order of the columns in each of the queries making up the UNION query must be the same.
3. The data types of the columns in each of the queries must be compatible.

Note that there is nothing in those rules about the names of the columns in the queries that make up the UNION query.

Listing 3.9 shows how to combine all of the data into a normalized view.

Listing 3.9 Using UNION to normalize the data

```
SELECT Category, OctQuantity, OctSales
FROM SalesSummary
UNION
SELECT Category, NovQuantity, NovSales
FROM SalesSummary
UNION
SELECT Category, DecQuantity, DecSales
FROM SalesSummary
UNION
SELECT Category, JanQuantity, JanSales
FROM SalesSummary
UNION
SELECT Category, FebQuantity, FebSales
FROM SalesSummary;
```

Table 3.2 shows a partial extract of the data returned.

Table 3.2 Partial extract of data returned by the UNION query in Listing 3.9

Category	OctQuantity	OctSales
Accessories	923	60883.03
Accessories	930	61165.40
. . .	. . .	. . .
Bikes	450	585130.50
Bikes	542	705733.50

continues

Table 3.2 Partial extract of data returned by the UNION query in Listing 3.9 (*continued*)

Category	OctQuantity	OctSales
Car racks	96	16772.05
Car racks	115	20137.05
Car racks	124	21763.30
. . .	. . .	. . .
Skateboards	203	89040.58
Skateboards	204	79461.30
Tires	110	3081.24
Tires	137	3937.70
Tires	150	4388.55
Tires	151	4356.91
Tires	186	5377.60

Two things should stand out. First, there is no way to distinguish to which month the data applies. The first two rows, for instance, represent the quantity and sales amount for Accessories for October and November, but there is no way to tell that from the data. As well, despite the fact that the data represents five months of sales, the columns are named OctQuantity and OctSales. That is because UNION queries get their column names from the names of the columns in the first SELECT statement.

Listing 3.10 shows a query that remedies both of those issues.

Listing 3.10 Tidying up the UNION query used to normalize the data

```
SELECT Category, 'Oct' AS SalesMonth, OctQuantity AS Quantity,
  OctSales AS SalesAmt
FROM SalesSummary
UNION
SELECT Category, 'Nov', NovQuantity, NovSales
FROM SalesSummary
UNION
SELECT Category, 'Dec', DecQuantity, DecSales
FROM SalesSummary
```

```
UNION
SELECT Category, 'Jan', JanQuantity, JanSales
FROM SalesSummary
UNION
SELECT Category, 'Feb', FebQuantity, FebSales
FROM SalesSummary;
```

Table 3.3 shows the same partial extract returned by the query in Listing 3.10.

Table 3.3 Partial extract of data returned by the UNION query in Listing 3.10

Category	SalesMonth	Quantity	SalesAmount
Accessories	Dec	987	62758.14
Accessories	Feb	979	60242.47
. . .	. . .	. . .	. . .
Bikes	Nov	412	546657.00
Bikes	Oct	413	536590.50
Car racks	Dec	115	20137.05
Car racks	Feb	124	21763.30
Car racks	Jan	142	24794.75
. . .	. . .	. . .	. . .
Skateboards	Nov	203	89040.58
Skateboards	Oct	164	60530.06
Tires	Dec	150	4388.55
Tires	Feb	137	3937.70
Tires	Jan	186	5377.60
Tires	Nov	110	3081.24
Tires	Oct	151	4356.91

Should you want the data presented in a different sequence, the ORDER BY clause must appear after the last SELECT in the UNION query, as shown in Listing 3.11 on the next page.

Listing 3.11 Specifying the sort order of the UNION query

```
SELECT Category, 'Oct' AS SalesMonth, OctQuantity AS Quantity,
  OctSales AS SalesAmt
FROM SalesSummary
UNION
SELECT Category, 'Nov', NovQuantity, NovSales
FROM SalesSummary
UNION
SELECT Category, 'Dec', DecQuantity, DecSales
FROM SalesSummary
UNION
SELECT Category, 'Jan', JanQuantity, JanSales
FROM SalesSummary
UNION
SELECT Category, 'Feb', FebQuantity, FebSales
FROM SalesSummary
ORDER BY SalesMonth, Category;
```

Table 3.4 shows a partial extract returned by the query in Listing 3.11.

Table 3.4 Partial extract of data returned by the UNION query in Listing 3.11

Category	SalesMonth	Quantity	SalesAmount
Accessories	Dec	987	62758.14
Bikes	Dec	332	439831.50
Car racks	Dec	115	20137.05
Clothing	Dec	139	4937.74
Components	Dec	265	27480.22
Skateboards	Dec	129	59377.20
Tires	Dec	150	4388.55
Accessories	Feb	979	60242.47
Bikes	Feb	450	585130.50
Car racks	Feb	124	21763.30
. . .	. . .	. . .	. . .

Note

Some DBMSs (such as Microsoft Access) allow you to put `ORDER BY` clauses other than at the end, but they do not actually cause the order to change.

When specifying the columns in the `ORDER BY` clause, usually you have the option of referring to them by name (remembering that the column names are specified in the first SELECT) or by position number. In other words, Listing 3.11 could use `ORDER BY 2, 1` instead of `ORDER BY SalesMonth, Category`. Oracle, however, insists on using ordinal references.

Another consideration is that `UNION` queries eliminate any duplicate rows. Should this not be what you want, you can specify `UNION ALL` instead of `UNION`, and duplicates will not be eliminated. On the other hand, `UNION ALL` can provide performance improvements as it skips the step of deduplicating the result set, so if you know that the sources will not overlap, it can be advantageous to specify `UNION ALL` for those queries.

Things to Remember

- Each of the `SELECT` statements in the `UNION` query must have the same number of columns.
- Although the names of the columns in the various `SELECT` statements do not matter, the data types of each column must be compatible.
- To control the order in which the data appears, you can use an `ORDER BY` clause after the last `SELECT` statement.
- Use `UNION ALL` rather than `UNION` if you do not wish to eliminate duplicate rows or pay the performance penalty of deduplicating rows.

This page intentionally left blank

Index

Symbols

A

B

I

W

X

Z