Microsoft

Microsoft® SQL Server® 2012 T-SQL Fundamentals

Itzik Ben-Gan

SolidQ

Microsoft® SQL Server® 2012 T-SQL Fundamentals

Gain a solid understanding of T-SQL—and write better queries

Master the fundamentals of Transact-SQL—and develop your own code for querying and modifying data in Microsoft SQL Server 2012. Led by a SQL Server expert, you'll learn the concepts behind T-SQL querying and programming, and then apply your knowledge with exercises in each chapter. Once you understand the logic behind T-SQL, you'll quickly learn how to write effective code—whether you're a programmer or database administrator.

Discover how to:

- Work with programming practices unique to T-SQL
- Create database tables and define data integrity
- Query multiple tables using joins and subqueries
- Simplify code and improve maintainability with table expressions
- Implement insert, update, delete, and merge data modification strategies
- Tackle advanced techniques such as window functions, pivoting, and grouping sets
- Control data consistency using isolation levels, and mitigate deadlocks and blocking
- Take T-SQL to the next level with programmable objects

Get code samples on the web
Ready to download at
http://go.microsoft.com/FWLink/?Linkid=248717

*For **system requirements**, see the Introduction.*

About the Author

Itzik Ben-Gan, Microsoft MVP for SQL Server since 1999, is cofounder of SolidQ, where he teaches and consults internationally on T-SQL querying, programming, and query tuning. He's a frequent contributor to *SQL Server Pro* and *MSDN Magazine*, and speaks at industry events such as Microsoft TechEd, DevTeach, PASS, and SQL Server Connections.

DEVELOPER ROADMAP

Start Here!
- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects

Step by Step
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook

Developer Reference
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples

Focused Topics
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples

microsoft.com/mspress

ISBN: 978-0-7356-58141

U.S.A. $49.99
Canada $52.99
[*Recommended*]

Databases/Microsoft SQL Server

Microsoft

Microsoft®

Microsoft® SQL Server® 2012 T-SQL Fundamentals

Itzik Ben-Gan

ISBN: 978-0-735-65814-1

Fifth Printing: March 2015

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at *mspinput@microsoft.com*. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Kristen Borg

Editorial Production and Illustration: Online Training Solutions, Inc.

Technical Reviewer: Gianluca Hotz and Herbert Albert

Copyeditor: Kathy Krause

Indexer: Allegro Technical Indexing

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

To Dato

To live in hearts we leave behind,
Is not to die.

—Thomas Campbell

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

I'm very happy that Itzik has managed to find the time and energy to produce a book about T-SQL fundamentals. For many years, Itzik has been using his great Microsoft SQL Server teaching, mentoring, and consulting experience to write books on *advanced* programming subjects, leaving a significant gap not only for the novice and less experienced users but also for the many experts working with SQL Server in roles where T-SQL programming is not a high priority.

When it comes to T-SQL, Itzik is one of the most knowledgeable people in the world. In fact, we (members of the SQL Server development team), turn to Itzik for expert advice on most of the new language extensions we plan to implement. His feedback and consultations have become an important part of our SQL Server development process.

It is never an easy task for a person who is a subject matter expert to write an introductory book; however, Itzik has the advantage of having taught both introductory and advanced programming classes for many years. Such experience is a great asset when differentiating the fundamental T-SQL information from the more advanced topics. But in this book, Itzik is not simply avoiding anything considered *advanced*; he is not afraid to take on inherently complex subjects such as set theory, predicate logic, and the relational model, introducing them in simple terms, and providing just enough information for readers to understand their importance to the SQL language. The result is a book that rewards readers with an understanding of not only *what* and *how* T-SQL works, but also *why*.

In programming manuals and books, there is no better way to convey the subject under discussion than with a good example. This book includes many examples—and you can download them all from Itzik's website, *http://tsql.solidq.com*. T-SQL is a dialect of the official ISO and ANSI standards for the SQL language, but it has numerous extensions that can improve the expressiveness and brevity of your T-SQL code. Many of Itzik's examples show the T-SQL dialect solution and the equivalent ANSI SQL solution to the same exercise side by side. This is a great advantage for readers who are familiar with the ANSI version of SQL but who are new to T-SQL, as well as for programmers who need to write SQL code that can be deployed easily across several different database platforms.

Itzik's deep connection to the SQL Server team shows in his explanation of the Appliance, Box, Cloud (ABC) flavors of SQL Server in Chapter 1, "Background to T-SQL Querying and Programming." So far, I have seen the term "ABC" used only internally within the Microsoft SQL Server team, but I'm sure it is only a matter of time until the term spreads around. Itzik developed and tested the examples in the book against both the "B" (box) and "C" (cloud) flavors of SQL Server. And the Appendix points out where you can get started with the cloud version of SQL Server, known as Windows Azure SQL Database. Therefore, you can use this book as a starting point for your own cloud experiences. The Azure website shows how to start your free subscription to the Azure services, so you can then execute the examples in the book.

The cloud extension of SQL Server is an extremely important point that you should not miss. I consider it to be *so* important that I'm doing something here that never should be done in a Foreword—advertising another book (sorry, Itzik, I have to do this!). My own interest and belief in cloud computing skyrocketed after reading Nicholas G. Carr's *The Big Switch* (W.W. Norton and Company, 2009), and I want to share that experience. It is a great book that compares the advancement of cloud computing to electrification in the early 1900s. My certainty in the future of cloud computing was further cemented by watching James Hamilton's "Cloud Computing Economies of Scale" presentation at the MIX10 conference (the recording is available at *http://channel9.msdn.com/events/MIX/MIX10/EX01*).

Itzik mentions one more cloud-related change that you should be aware of. We were used to multi-year gaps between SQL Server releases, but that pattern is changing significantly with the cloud; you should instead be prepared for several smaller cloud releases (called Service Updates) deployed in the Microsoft Data Centers around the world every year. Therefore, Itzik wisely documents the discrepancies between SQL Server and Windows Azure SQL Database T-SQL on his *http://tsql.solidq.com* website rather than in the book, so he can easily keep the information up to date.

Enjoy the book—and even more—enjoy the new insights into T-SQL that this book will bring to you.

Lubor Kollar, SQL Server development team, Microsoft

Introduction

This book walks you through your first steps in T-SQL (also known as Transact-SQL), which is the Microsoft SQL Server dialect of the ISO and ANSI standards for SQL. You'll learn the theory behind T-SQL querying and programming and how to develop T-SQL code to query and modify data, and you'll get an overview of programmable objects.

Although this book is intended for beginners, it is not merely a set of procedures for readers to follow. It goes beyond the syntactical elements of T-SQL and explains the logic behind the language and its elements.

Occasionally, the book covers subjects that may be considered advanced for readers who are new to T-SQL; therefore, those sections are optional reading. If you already feel comfortable with the material discussed in the book up to that point, you might want to tackle the more advanced subjects; otherwise, feel free to skip those sections and return to them after you've gained more experience. The text will indicate when a section may be considered more advanced and is provided as optional reading.

Many aspects of SQL are unique to the language and are very different from other programming languages. This book helps you adopt the right state of mind and gain a true understanding of the language elements. You learn how to think in terms of sets and follow good SQL programming practices.

The book is not version-specific; it does, however, cover language elements that were introduced in recent versions of SQL Server, including SQL Server 2012. When I discuss language elements that were introduced recently, I specify the version in which they were added.

Besides being available in an on-premises flavor, SQL Server is also available as a cloud-based service called Windows Azure SQL Database (formerly called SQL Azure). The code samples in this book were tested against both on-premises SQL Server and SQL Database. The book's companion website (*http://tsql.solidq.com*) provides information about compatibility issues between the flavors—for example, features that are available in SQL Server 2012 but not yet in SQL Database.

To complement the learning experience, the book provides exercises that enable you to practice what you've learned. The book occasionally provides optional exercises that are more advanced. Those exercises are intended for readers who feel very comfortable with the material and want to challenge themselves with more difficult problems. The optional exercises for advanced readers are labeled as such.

Who Should Read This Book

This book is intended for T-SQL developers, DBAs, BI practitioners, report writers, analysts, architects, and SQL Server power users who just started working with SQL Server and need to write queries and develop code using Transact-SQL.

Assumptions

To get the most out of this book, you should have working experience with Windows and with applications based on Windows. You should also be familiar with basic concepts concerning relational database management systems.

Who Should Not Read This Book

Not every book is aimed at every possible audience. This book covers fundamentals. It is mainly aimed at T-SQL practitioners with little or no experience. With that said, several readers of the previous edition of this book have mentioned that—even though they already had years of experience—they still found the book useful for filling gaps in their knowledge.

Organization of This Book

This book starts with both a theoretical background to T-SQL querying and programming in Chapter 1, laying the foundations for the rest of the book, and also coverage of creating tables and defining data integrity. The book moves on to various aspects of querying and modifying data in Chapters 2 through 8, then to a discussion of concurrency and transactions in Chapter 9, and finally provides an overview of programmable objects in Chapter 10. The following section lists the chapter titles along with a short description:

- Chapter 1, "Background to T-SQL Querying and Programming," provides a theoretical background of SQL, set theory, and predicate logic; examines the relational model and more; describes SQL Server's architecture; and explains how to create tables and define data integrity.
- Chapter 2, "Single-Table Queries," covers various aspects of querying a single table by using the *SELECT* statement.

- Chapter 3, "Joins," covers querying multiple tables by using joins, including cross joins, inner joins, and outer joins.
- Chapter 4, "Subqueries," covers queries within queries, otherwise known as subqueries.
- Chapter 5, "Table Expressions," covers derived tables, common table expressions (CTEs), views, inline table-valued functions, and the *APPLY* operator.
- Chapter 6, "Set Operators," covers the set operators *UNION*, *INTERSECT*, and *EXCEPT*.
- Chapter 7, "Beyond the Fundamentals of Querying," covers window functions, pivoting, unpivoting, and working with grouping sets.
- Chapter 8, "Data Modification," covers inserting, updating, deleting, and merging data.
- Chapter 9, "Transactions and Concurrency," covers concurrency of user connections that work with the same data simultaneously; it covers concepts including transactions, locks, blocking, isolation levels, and deadlocks.
- Chapter 10, "Programmable Objects," provides an overview of the T-SQL programming capabilities in SQL Server.
- The book also provides an appendix, "Getting Started," to help you set up your environment, download the book's source code, install the *TSQL2012* sample database, start writing code against SQL Server, and learn how to get help by working with SQL Server Books Online.

System Requirements

The Appendix, "Getting Started," explains which editions of SQL Server 2012 you can use to work with the code samples included with this book. Each edition of SQL Server might have different hardware and software requirements, and those requirements are well documented in SQL Server Books Online under "Hardware and Software Requirements for Installing SQL Server 2012." The Appendix also explains how to work with SQL Server Books Online.

If you're connecting to SQL Database, hardware and server software are handled by Microsoft, so those requirements are irrelevant in this case.

Code Samples

This book features a companion website that makes available to you all the code used in the book, the errata, and additional resources.

http://tsql.solidq.com

Refer to the Appendix, "Getting Started," for details about the source code.

Acknowledgments

Many people contributed to making this book a reality, whether directly or indirectly, and deserve thanks and recognition.

To Lilach, for giving reason to everything I do, and for not complaining about the endless hours I spend on SQL.

To my parents Mila and Gabi and to my siblings Mickey and Ina, thanks for the constant support. Thanks for accepting the fact that I'm away, which is now harder than ever. Mom, we're all counting on you to be well and are encouraged by your strength and determination. Dad, thanks for being so supportive.

To members of the Microsoft SQL Server development team; Lubor Kollar, Tobias Ternstrom, Umachandar Jayachandran (UC), and I'm sure many others. Thanks for the great effort, and thanks for all the time you spent meeting me and responding to my email messages, addressing my questions and requests for clarification. I think that SQL Server 2012 and SQL Database show great investment in T-SQL, and I hope this will continue.

To the editorial team at O'Reilly Media and Microsoft Press; to Ken Jones, thanks for all the Itzik hours you spent, and thanks for initiating the project. To Russell Jones, thanks for your efforts in taking over the project and running it from the O'Reilly side. Also thanks to Kristen Borg, Kathy Krause, and all others who worked on the book.

To Herbert Albert and Gianluca Hotz, thanks for your work as the technical editors of the book. Your edits were excellent and I'm sure they improved the book's quality and accuracy.

To SolidQ, my company for the last decade: it's gratifying to be part of such a great company that evolved to what it is today. The members of this company are much more than colleagues to me; they are partners, friends, and family. Thanks to Fernando G. Guerrero, Douglas McDowell, Herbert Albert, Dejan Sarka, Gianluca Hotz, Jeanne Reeves,

Glenn McCoin, Fritz Lechnitz, Eric Van Soldt, Joelle Budd, Jan Taylor, Marilyn Templeton, Berry Walker, Alberto Martin, Lorena Jimenez, Ron Talmage, Andy Kelly, Rushabh Mehta, Eladio Rincón, Erik Veerman, Jay Hackney, Richard Waymire, Carl Rabeler, Chris Randall, Johan Åhlén, Raoul Illyés, Peter Larsson, Peter Myers, Paul Turley, and so many others.

To members of the *SQL Server Pro* editorial team, Megan Keller, Lavon Peters, Michele Crockett, Mike Otey, and I'm sure many others; I've been writing for the magazine for more than a decade and am grateful for the opportunity to share my knowledge with the magazine's readers.

To SQL Server MVPs Alejandro Mesa, Erland Sommarskog, Aaron Bertrand, Tibor Karaszi, Paul White, and many others, and to the MVP lead, Simon Tien; this is a great program that I'm grateful and proud to be part of. The level of expertise of this group is amazing and I'm always excited when we all get to meet, both to share ideas and just to catch up at a personal level over beer. I believe that, in great part, Microsoft's inspiration to add new T-SQL capabilities in SQL Server is thanks to the efforts of SQL Server MVPs, and more generally the SQL Server community. It is great to see this synergy yielding such a meaningful and important outcome.

To Q2, Q3, and Q4, thanQ.

Finally, to my students: teaching SQL is what drives me. It's my passion. Thanks for allowing me to fulfill my calling, and for all the great questions that make me seek more knowledge.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735658141

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

CHAPTER 1

Background to T-SQL Querying and Programming

You're about to embark on a journey to a land that is like no other—a land that has its own set of laws. If reading this book is your first step in learning Transact-SQL (T-SQL), you should feel like Alice—just before she started her adventures in Wonderland. For me, the journey has not ended; instead, it's an ongoing path filled with new discoveries. I envy you; some of the most exciting discoveries are still ahead of you!

I've been involved with T-SQL for many years: teaching, speaking, writing, and consulting about it. For me, T-SQL is more than just a language—it's a way of thinking. I've taught and written extensively on advanced topics, but until now, I have postponed writing about fundamentals. This is not because T-SQL fundamentals are simple or easy—in fact, just the opposite: The apparent simplicity of the language is misleading. I could explain the language syntax elements in a superficial manner and have you writing queries within minutes. But that approach would only hold you back in the long run and make it harder for you to understand the essence of the language.

Acting as your guide while you take your first steps in this realm is a big responsibility. I wanted to make sure that I spent enough time and effort exploring and understanding the language before writing about fundamentals. T-SQL is deep; learning the fundamentals the right way involves much more than just understanding the syntax elements and coding a query that returns the right output. You pretty much need to forget what you know about other programming languages and start thinking in terms of T-SQL.

Theoretical Background

SQL stands for *Structured Query Language*. SQL is a standard language that was designed to query and manage data in relational database management systems (RDBMSs). An RDBMS is a database management system based on the relational model (a semantic model for representing data), which in turn is based on two mathematical branches: set theory and predicate logic. Many other programming languages and various aspects of computing evolved pretty much as a result of intuition. In contrast, to the degree that SQL is based on the relational model, it is based on a firm foundation—applied mathematics. T-SQL thus sits on wide and solid shoulders. Microsoft provides T-SQL as a dialect of, or extension to, SQL in Microsoft SQL Server data management software, its RDBMS.

This section provides a brief theoretical background about SQL, set theory and predicate logic, the relational model, and the data life cycle. Because this book is neither a mathematics book nor a design/data modeling book, the theoretical information provided here is informal and by no means complete. The goals are to give you a context for the T-SQL language and to deliver the key points that are integral to correctly understanding T-SQL later in the book.

Language Independence

The relational model is language-independent. That is, you can implement the relational model with languages other than SQL—for example, with C# in a class model. Today it is common to see RDBMSs that support languages other than a dialect of SQL, such as the CLR integration in SQL Server.

Also, you should realize from the start that SQL deviates from the relational model in several ways. Some even say that a new language—one that more closely follows the relational model—should replace SQL. But to date, SQL is the industrial language used by all leading RDBMSs in practice.

See Also *For details about the deviations of SQL from the relational model, as well as how to use SQL in a relational way, see this book on the topic:* SQL and Relational Theory: How to Write Accurate SQL Code, Second Edition *by C. J. Date (O'Reilly Media, 2011).*

SQL

SQL is both an ANSI and ISO standard language based on the relational model, designed for querying and managing data in an RDBMS.

In the early 1970s, IBM developed a language called SEQUEL (short for Structured English QUEry Language) for their RDBMS product called System R. The name of the language was later changed from SEQUEL to SQL because of a trademark dispute. SQL first became an ANSI standard in 1986, and then an ISO standard in 1987. Since 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) have been releasing revisions for the SQL standard every few years. So far, the following standards have been released: SQL-86 (1986), SQL-89 (1989), SQL-92 (1992), SQL:1999 (1999), SQL:2003 (2003), SQL:2006 (2006), SQL:2008 (2008), and SQL:2011 (2011).

Interestingly, SQL resembles English and is also very logical. Unlike many programming languages, which use an imperative programming paradigm, SQL uses a declarative one. That is, SQL requires you to specify *what* you want to get and not *how* to get it, letting the RDBMS figure out the physical mechanics required to process your request.

SQL has several categories of statements, including Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL). DDL deals with object definitions and includes statements such as *CREATE, ALTER,* and *DROP*. DML allows you to query and modify data and includes statements such as *SELECT, INSERT, UPDATE, DELETE, TRUNCATE,* and *MERGE*. It's a

common misunderstanding that DML includes only data modification statements, but as I mentioned, it also includes *SELECT*. Another common misunderstanding is that *TRUNCATE* is a DDL statement, but in fact it is a DML statement. DCL deals with permissions and includes statements such as *GRANT* and *REVOKE*. This book focuses on DML.

T-SQL is based on standard SQL, but it also provides some nonstandard/proprietary extensions. When describing a language element for the first time, I'll typically mention whether it is standard.

Set Theory

Set theory, which originated with the mathematician Georg Cantor, is one of the mathematical branches on which the relational model is based. Cantor's definition of a set follows:

> *By a "set" we mean any collection M into a whole of definite, distinct objects m (which are called the "elements" of M) of our perception or of our thought.*
>
> *—Joseph W. Dauben and Georg Cantor (Princeton University Press, 1990)*

Every word in the definition has a deep and crucial meaning. The definitions of a set and set membership are axioms that are not supported by proofs. Each element belongs to a universe, and either is or is not a member of the set.

Let's start with the word *whole* in Cantor's definition. A set should be considered a single entity. Your focus should be on the collection of objects as opposed to the individual objects that make up the collection. Later on, when you write T-SQL queries against tables in a database (such as a table of employees), you should think of the set of employees as a whole rather than the individual employees. This might sound trivial and simple enough, but apparently many programmers have difficulty adopting this way of thinking.

The word *distinct* means that every element of a set must be unique. Jumping ahead to tables in a database, you can enforce the uniqueness of rows in a table by defining key constraints. Without a key, you won't be able to uniquely identify rows, and therefore the table won't qualify as a set. Rather, the table would be a *multiset* or a *bag*.

The phrase *of our perception or of our thought* implies that the definition of a set is subjective. Consider a classroom: One person might perceive a set of people, whereas another might perceive a set of students and a set of teachers. Therefore, you have a substantial amount of freedom in defining sets. When you design a data model for your database, the design process should carefully consider the subjective needs of the application to determine adequate definitions for the entities involved.

As for the word *object*, the definition of a set is not restricted to physical objects such as cars or employees but rather is relevant to abstract objects as well, such as prime numbers or lines.

What Cantor's definition of a set leaves out is probably as important as what it includes. Notice that the definition doesn't mention any order among the set elements. The order in which set elements are listed is not important. The formal notation for listing set elements uses curly brackets: {a, b, c}. Because order has no relevance, you can express the same set as *{b, a, c}* or *{b, c, a}*. Jumping

ahead to the set of attributes (called *columns* in SQL) that make up the header of a relation (called a *table* in SQL), an element is supposed to be identified by name—not by ordinal position.

Similarly, consider the set of tuples (called *rows* by SQL) that make up the body of the relation; an element is identified by its key values—not by position. Many programmers have a hard time adapting to the idea that, with respect to querying tables, there is no order among the rows. In other words, a query against a table can return table rows in *any order* unless you explicitly request that the data be sorted in a specific way, perhaps for presentation purposes.

Predicate Logic

Predicate logic, whose roots reach back to ancient Greece, is another branch of mathematics on which the relational model is based. Dr. Edgar F. Codd, in creating the relational model, had the insight to connect predicate logic to both management and querying of data. Loosely speaking, a *predicate* is a property or an expression that either holds or doesn't hold—in other words, is either true or false. The relational model relies on predicates to maintain the logical integrity of the data and define its structure. One example of a predicate used to enforce integrity is a constraint defined in a table called *Employees* that allows only employees with a salary greater than zero to be stored in the table. The predicate is "salary greater than zero" (T-SQL expression: *salary > 0*).

You can also use predicates when filtering data to define subsets, and more. For example, if you need to query the *Employees* table and return only rows for employees from the sales department, you would use the predicate "department equals sales" in your query filter (T-SQL expression: *department = 'sales'*).

In set theory, you can use predicates to define sets. This is helpful because you can't always define a set by listing all its elements (for example, infinite sets), and sometimes for brevity it's more convenient to define a set based on a property. As an example of an infinite set defined with a predicate, the set of all prime numbers can be defined with the following predicate: "*x* is a positive integer greater than 1 that is divisible only by 1 and itself." For any specified value, the predicate is either true or not true. The set of all prime numbers is the set of all elements for which the predicate is true. As an example of a finite set defined with a predicate, the set *{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}* can be defined as the set of all elements for which the following predicate holds true: "*x* is an integer greater than or equal to 0 and smaller than or equal to 9."

The Relational Model

The relational model is a semantic model for data management and manipulation and is based on set theory and predicate logic. As mentioned earlier, it was created by Dr. Edgar F. Codd, and later explained and developed by Chris Date, Hugh Darwen, and others. The first version of the relational model was proposed by Codd in 1969 in an IBM research report called "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks." A revised version was proposed by Codd in 1970 in a paper called "A Relational Model of Data for Large Shared Data Banks," published in the journal *Communications of the ACM*.

The goal of the relational model is to enable consistent representation of data with minimal or no redundancy and without sacrificing completeness, and to define data integrity (enforcement of data consistency) as part of the model. An RDBMS is supposed to implement the relational model and provide the means to store, manage, enforce the integrity of, and query data. The fact that the relational model is based on a strong mathematical foundation means that given a certain data model instance (from which a physical database will later be generated), you can tell with certainty when a design is flawed, rather than relying solely on intuition.

The relational model involves concepts such as propositions, predicates, relations, tuples, attributes, and more. For non-mathematicians, these concepts can be quite intimidating. The sections that follow cover some of the key aspects of the model in an informal, nonmathematical manner and explain how they relate to databases.

Propositions, Predicates, and Relations

The common belief that the term *relational* stems from relationships between tables is incorrect. "Relational" actually pertains to the mathematical term *relation*. In set theory, a relation is a representation of a set. In the relational model, a relation is a set of related information, with the counterpart in SQL being a table—albeit not an exact counterpart. A key point in the relational model is that a single relation should represent a single set (for example, *Customers*). It is interesting to note that operations on relations (based on relational algebra) result in a relation (for example, a join between two relations).

Note The relational model distinguishes between a *relation* and a *relation variable*, but to keep things simple, I won't get into this distinction; instead, I'll use the term *relation* for both cases. Also, a relation is made of a header and a body. The header consists of a set of attributes (called *columns* in SQL), where each element is identified by an attribute name and a type name. The body consists of a set of tuples (called *rows* in SQL), where each element is identified by a key. To keep things simple, I'll refer to a table as a set of rows.

When you design a data model for a database, you represent all data with relations (tables). You start by identifying propositions that you will need to represent in your database. A proposition is an assertion or a statement that must be true or false. For example, the statement, "Employee Itzik Ben-Gan was born on February 12, 1971, and works in the IT department" is a proposition. If this proposition is true, it will manifest itself as a row in a table of *Employees*. A false proposition simply won't manifest itself. This presumption is known as the *close world assumption (CWA)*.

The next step is to formalize the propositions. You do this by taking out the actual data (the body of the relation) and defining the structure (the heading of the relation)—for example, by creating predicates out of propositions. You can think of predicates as parameterized propositions. The heading of a relation comprises a set of attributes. Note the use of the term "set"; in the relational model, attributes are unordered and distinct. An attribute is identified by an attribute name and a type name. For example, the heading of an *Employees* relation might consist of the following attributes (expressed as pairs of attribute names and type names): *employeeid* integer, *firstname* character string, *lastname* character string, *birthdate* date, *departmentid* integer.

A type is one of the most fundamental building blocks for relations. A type constrains an attribute to a certain set of possible or valid values. For example, the type *INT* is the set of all integers in the range –2,147,483,648 to 2,147,483,647. A type is one of the simplest forms of a predicate in a database because it restricts the attribute values that are allowed. For example, the database would not accept a proposition where an employee birth date is February 31, 1971 (not to mention a birth date stated as something like "abc!"). Note that types are not restricted to base types such as integers or character strings; a type could also be an enumeration of possible values, such as an enumeration of possible job positions. A type can be complex. Probably the best way to think of a type is as a class—encapsulated data and the behavior supporting it. An example of a complex type would be a geometry type that supports polygons.

Missing Values

One aspect of the relational model is the source of many passionate debates—whether predicates should be restricted to two-valued logic. That is, in two-valued predicate logic, a predicate is either true or false. If a predicate is not true, it must be false. Use of two-valued predicate logic follows a mathematical law called the law of excluded middle. However, some say that there's room for three-valued (or even four-valued) predicate logic, taking into account cases where values are missing. A predicate involving a missing value yields neither true nor false—it yields unknown. Take, for example, a mobile phone attribute of an *Employees* relation. Suppose that a certain employee's mobile phone number is missing. How do you represent this fact in the database? In a three-valued logic implementation, the mobile phone attribute should allow a special mark for a missing value. Then a predicate comparing the mobile phone attribute with some specific number will yield *unknown* for the case with the missing value. Three-valued predicate logic refers to the three possible logical values that can result from a predicate—*true*, *false*, and *unknown*.

Some people believe that three-valued predicate logic is non-relational, whereas others believe that it is relational. Codd actually advocated four-valued predicate logic, saying that there were two different cases of missing values: missing but applicable (A-Mark), and missing but inapplicable (I-Mark). An example of "missing but applicable" is when an employee has a mobile phone, but you don't know what the mobile phone number is. An example of missing but inapplicable is when an employee doesn't have a mobile phone at all. According to Codd, two special markers should be used to support these two cases of missing values. SQL implements three-valued predicate logic by supporting the *NULL* mark to signify the generic concept of a missing value. Support for *NULL* marks and three-valued predicate logic in SQL is the source of a great deal of confusion and complexity, though one can argue that missing values are part of reality. In addition, the alternative—using only two-valued predicate logic—is no less problematic.

Constraints

One of the greatest benefits of the relational model is the ability to define data integrity as part of the model. Data integrity is achieved through rules called *constraints* that are defined in the data model and enforced by the RDBMS. The simplest methods of enforcing integrity are assigning an attribute type with its attendant "nullability" (whether it supports or doesn't support *NULL* marks). Constraints are also enforced through the model itself; for example, the relation *Orders(orderid, orderdate,*

duedate, shipdate) allows three distinct dates per order, whereas the relations *Employees(empid)* and *EmployeeChildren(empid, childname)* allow zero to countable infinity children per employee.

Other examples of constraints include *candidate keys,* which provide entity integrity, and *foreign keys,* which provide referential integrity. A candidate key is a key defined on one or more attributes that prevents more than one occurrence of the same tuple (*row* in SQL) in a relation. A predicate based on a candidate key can uniquely identify a row (such as an employee). You can define multiple candidate keys in a relation. For example, in an *Employees* relation, you can define candidate keys on *employeeid*, on *SSN* (Social Security number), and others. Typically, you arbitrarily choose one of the candidate keys as the *primary key* (for example, *employeeid* in the *Employees* relation), and use that as the preferred way to identify a row. All other candidate keys are known as *alternate keys*.

Foreign keys are used to enforce referential integrity. A foreign key is defined on one or more attributes of a relation (known as the *referencing relation*) and references a candidate key in another (or possibly the same) relation. This constraint restricts the values in the referencing relation's foreign key attributes to the values that appear in the referenced relation's candidate key attributes. For example, suppose that the *Employees* relation has a foreign key defined on the attribute *departmentid*, which references the primary key attribute *departmentid* in the *Departments* relation. This means that the values in *Employees.departmentid* are restricted to the values that appear in *Departments.departmentid*.

Normalization

The relational model also defines *normalization rules* (also known as *normal forms*). Normalization is a formal mathematical process to guarantee that each entity will be represented by a single relation. In a normalized database, you avoid anomalies during data modification and keep redundancy to a minimum without sacrificing completeness. If you follow Entity Relationship Modeling (ERM), and represent each entity and its attributes, you probably won't need normalization; instead, you will apply normalization only to reinforce and ensure that the model is correct. The following sections briefly cover the first three normal forms (1NF, 2NF, and 3NF) introduced by Codd.

1NF The first normal form says that the tuples (rows) in the relation (table) must be unique, and attributes should be atomic. This is a redundant definition of a relation; in other words, if a table truly represents a relation, it is already in first normal form.

You achieve unique rows by defining a unique key for the table.

You can only operate on attributes with operations that are defined as part of the attribute's type. Atomicity of attributes is subjective in the same way that the definition of a set is subjective. As an example, should an employee name in an *Employees* relation be expressed with one attribute (*fullname*), two (*firstname* and *lastname*), or three (*firstname*, *middlename*, and *lastname*)? The answer depends on the application. If the application needs to manipulate the parts of the employee's name separately (such as for search purposes), it makes sense to break them apart; otherwise, it doesn't.

In the same way that an attribute might not be atomic enough based on the needs of the application, an attribute might also be subatomic. For example, if an address attribute is considered atomic for a particular application, not including the city as part of the address would violate the first normal form.

This normal form is often misunderstood. Some people think that an attempt to mimic arrays violates the first normal form. An example would be defining a *YearlySales* relation with the following attributes: *salesperson*, *qty2010*, *qty2011*, and *qty2012*. However, in this example, you don't really violate the first normal form; you simply impose a constraint—restricting the data to three specific years: 2010, 2011, and 2012.

2NF The second normal form involves two rules. One rule is that the data must meet the first normal form. The other rule addresses the relationship between non-key and candidate key attributes. For every candidate key, every non-key attribute has to be fully functionally dependent on the entire candidate key. In other words, a non-key attribute cannot be fully functionally dependent on part of a candidate key. To put it more informally, if you need to obtain any non-key attribute value, you need to provide the values of all attributes of a candidate key from the same tuple. You can find any value of any attribute of any tuple if you know all the attribute values of a candidate key.

As an example of violating the second normal form, suppose that you define a relation called *Orders* that represents information about orders and order lines (see Figure 1-1). The *Orders* relation contains the following attributes: *orderid*, *productid*, *orderdate*, *qty*, *customerid*, and *companyname*. The primary key is defined on *orderid* and *productid*.

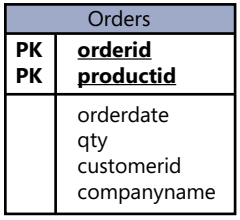

FIGURE 1-1 Data model before applying 2NF.

The second normal form is violated in Figure 1-1 because there are non-key attributes that depend on only part of a candidate key (the primary key, in this example). For example, you can find the *orderdate* of an order, as well as *customerid* and *companyname*, based on the *orderid* alone. To conform to the second normal form, you would need to split your original relation into two relations: *Orders* and *OrderDetails* (as shown in Figure 1-2). The *Orders* relation would include the attributes *orderid*, *orderdate*, *customerid*, and *companyname*, with the primary key defined on *orderid*. The *OrderDetails* relation would include the attributes *orderid*, *productid*, and *qty*, with the primary key defined on *orderid* and *productid*.

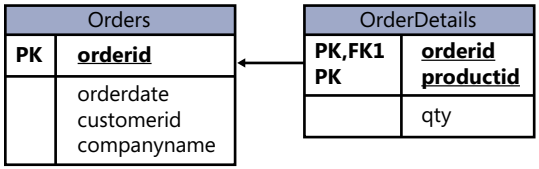

FIGURE 1-2 Data model after applying 2NF and before 3NF.

3NF The third normal form also has two rules. The data must meet the second normal form. Also, all non-key attributes must be dependent on candidate keys non-transitively. Informally this rule means

that all non-key attributes must be mutually independent. In other words, one non-key attribute cannot be dependent on another non-key attribute.

The *Orders* and *OrderDetails* relations described previously now conform to the second normal form. Remember that the *Orders* relation at this point contains the attributes *orderid*, *orderdate*, *customerid*, and *companyname*, with the primary key defined on *orderid*. Both *customerid* and *companyname* depend on the whole primary key—*orderid*. For example, you need the entire primary key to find the *customerid* representing the customer who placed the order. Similarly, you need the whole primary key to find the company name of the customer who placed the order. However, *customerid* and *companyname* are also dependent on each other. To meet the third normal form, you need to add a *Customers* relation (shown in Figure 1-3) with the attributes *customerid* (as the primary key) and *companyname*. Then you can remove the *companyname* attribute from the *Orders* relation.

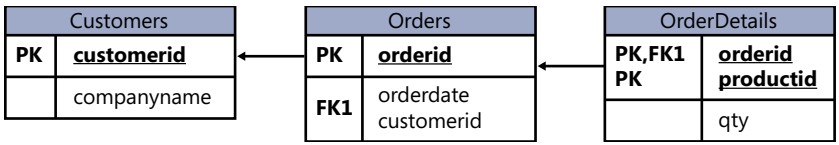

FIGURE 1-3 Data model after applying 3NF.

Informally, 2NF and 3NF are commonly summarized with the sentence, "Every non-key attribute is dependent on the key, the whole key, and nothing but the key—so help me Codd."

There are higher normal forms beyond Codd's original first three normal forms that involve compound primary keys and temporal databases, but they are outside the scope of this book.

The Data Life Cycle

Data is usually perceived as something static that is entered into a database and later queried. But in many environments, data is actually more similar to a product in an assembly line, moving from one environment to another and undergoing transformations along the way. This section describes the different environments in which data can reside and the characteristics of both the data and the environment at each stage of the data life cycle. Figure 1-4 illustrates the data life cycle.

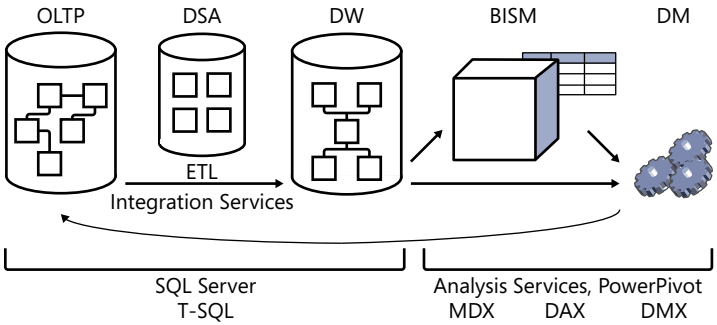

FIGURE 1-4 The data life cycle.

Here's a quick description of what each acronym represents:

- OLTP: online transactional processing
- DSA: data staging area
- DW: data warehouse
- BISM: Business Intelligence Semantic Model
- DM: data mining
- ETL: extract, transform, and load
- MDX: Multidimensional Expressions
- DAX: Data Analysis Expressions
- DMX: Data Mining Extensions

Online Transactional Processing

Data is entered initially into an online transactional processing (OLTP) system. The focus of an OLTP system is data entry and not reporting—transactions mainly insert, update, and delete data. The relational model is targeted primarily at OLTP systems, where a normalized model provides both good performance for data entry and data consistency. In a normalized environment, each table represents a single entity and keeps redundancy to a minimum. When you need to modify a fact, you need to modify it in only one place. This results in optimized performance for data modifications and little chance for error.

However, an OLTP environment is not suitable for reporting purposes because a normalized model usually involves many tables (one for each entity) with complex relationships. Even simple reports require joining many tables, resulting in complex and poorly performing queries.

You can implement an OLTP database in SQL Server and both manage it and query it with T-SQL.

Data Warehouses

A *data warehouse* (DW) is an environment designed for data retrieval and reporting purposes. When it serves an entire organization, such an environment is called a data warehouse; when it serves only part of the organization (such as a specific department) or a subject matter area in the organization, it is called a *data mart*. The data model of a data warehouse is designed and optimized mainly to support data retrieval needs. The model has intentional redundancy, fewer tables, and simpler relationships, ultimately resulting in simpler and more efficient queries as compared to an OLTP environment.

The simplest data warehouse design is called a *star schema*. The star schema includes several dimension tables and a fact table. Each dimension table represents a subject by which you want to analyze the data. For example, in a system that deals with orders and sales, you will probably want to analyze data by customers, products, employees, time, and similar subjects. In a star schema, each dimension is implemented as a single table with redundant data. For example, a product dimension

could be implemented as a single *ProductDim* table instead of three normalized tables: *Products*, *ProductSubCategories*, and *ProductCategories*. If you normalize a dimension table, which results in multiple tables representing that dimension, you get what's known as a *snowflake dimension*. A schema that contains snowflake dimensions is known as a *snowflake schema* (as opposed to a star schema).

The fact table holds the facts and measures such as quantity and value for each relevant combination of dimension keys. For example, for each relevant combination of customer, product, employee, and day, the fact table would have a row containing the quantity and value. Note that data in a data warehouse is typically preaggregated to a certain level of granularity (such as a day), unlike data in an OLTP environment, which is usually recorded at the transaction level.

Historically, early versions of SQL Server mainly targeted OLTP environments, but eventually SQL Server also started targeting data warehouse systems and data analysis needs. You can implement a data warehouse as a SQL Server database and manage and query it with T-SQL.

The process that pulls data from source systems (OLTP and others), manipulates it, and loads it into the data warehouse is called extract, transform, and load, or ETL. SQL Server provides a tool called Microsoft SQL Server Integration Services (SSIS) to handle ETL needs.

Often the ETL process will involve the use of a data staging area (DSA) between the OLTP and the DW. The DSA usually resides in a relational database such as a SQL Server database and is used as the data cleansing area. The DSA is not open to end users.

The Business Intelligence Semantic Model

The Business Intelligence Semantic Model (BISM) is Microsoft's latest model for supporting the entire BI stack of applications. The idea is to provide rich, flexible, efficient, and scalable analytical and reporting capabilities. Its architecture includes three layers: the data model, business logic and queries, and data access.

The deployment of the model can be in an Analysis Services server or PowerPivot. Analysis Services is targeted at BI professionals and IT, whereas PowerPivot is targeted at business users. With Analysis Services, you can use either a multidimensional data model or a tabular (relational) one. With PowerPivot, you use a tabular data model.

The business logic and queries use two languages: Multidimensional Expressions (MDX), based on multidimensional concepts, and Data Analysis Expressions (DAX), based on tabular concepts.

The data access layer can get its data from different sources: relational databases such as the DW, files, cloud services, line of business (LOB) applications, OData feeds, and others. The data access layer can either cache the data locally or just serve as a passthrough layer directly from the data sources. The cached mode can use one of two storage engines. One is a preaggregated form known as MOLAP that was originally designed to support the multidimensional model. Another is a newer engine called VertiPaq, which implements a columnstore concept, with very high levels of compression and a very fast processing engine, removing the need for preaggregations, indexing, and so on.

See Also *This section about BISM has a lot of concepts to digest—perhaps too many for a fundamentals book about T-SQL. If you are curious about BISM and would like a more detailed overview, you can find it in the following blog entry from the Analysis Services team:* http://blogs.msdn.com/b/analysisservices/archive/2011/05/16/analysis-services-vision-amp-roadmap-update.aspx.

Data Mining

BISM provides the user with answers to all possible questions, but the user's task is to ask the right questions—to sift anomalies, trends, and other useful information from the sea of data. In the dynamic analysis process, the user navigates from one view of aggregates to another—again, slicing and dicing the data—to find useful information.

Data mining (DM) is the next step; instead of letting the user look for useful information in the sea of data, data mining models can do this for the user. That is, data mining algorithms comb the data and sift the useful information from it. Data mining has enormous business value for organizations, helping to identify trends, figure out which products are purchased together, predict customer choices based on specific parameters, and more.

Analysis Services supports data mining algorithms—including clustering, decision trees, and others—to address such needs. The language used to manage and query data mining models is Data Mining Extensions (DMX).

SQL Server Architecture

This section will introduce you to the SQL Server architecture, the flavors of the product, the entities involved—SQL Server instances, databases, schemas, and database objects—and the purpose of each entity.

The ABC Flavors of SQL Server

For many years, SQL Server was available only in one flavor—a box, or on-premises, flavor. More recently, Microsoft decided to offer multiple flavors to allow customers to choose the one that best suits their needs. At the date of this writing, Microsoft provides three main flavors of SQL Server that are internally referred to as the *ABC flavors*: A for Appliance, B for Box, and C for Cloud.

Appliance

The idea behind the appliance flavor is to provide a complete solution including hardware, software, and services. The appliance is hosted locally at the customer site. There are several appliances available today, one of which is Parallel Data Warehouse (PDW). Microsoft partners with hardware vendors such as Dell and HP to provide the appliance offering. Experts from Microsoft and the hardware vendor handle the performance, security, and availability aspects for the customer.

This book's focus is T-SQL, so you are probably wondering which language is used to interact with the database engine. That depends on the appliance. For example, PDW doesn't use the same engine as the on-premises engine; it uses a specialized one. The specialized PDW engine uses its own flavor of SQL called *distributed SQL*, or DSQL. Microsoft's long-term goal is to align the language support in the different flavors of the product, but that has not yet been realized. This book focuses on T-SQL, which is supported by some of the appliances and the on-premises and cloud flavors.

Box

The box flavor of SQL Server, formally referred to as *on-premises SQL Server*, is the traditional one, usually installed on the customer's premises. The customer is responsible for everything—getting the hardware; installing the software; and handling updates, high availability and disaster recovery (HADR), security, and everything else.

The customer can install multiple instances of the product in the same server (more on this in the next section) and can write queries that interact with multiple databases. It is also possible to switch the connection between databases, unless one of them is a contained database.

The querying language used is T-SQL. You can run all of the code samples and exercises in this book on an on-premises SQL Server implementation, if you want. See the Appendix for details about obtaining and installing an evaluation edition of SQL Server, as well as creating the sample database.

Cloud

Microsoft supports two cloud flavors of SQL Server: private and public. The use of the term *cloud* for the private case could be a bit confusing, because it is hosted locally, but the private flavor uses virtualization technology. The engine is a box engine (hence the same T-SQL is used to query it), but it is limited by the virtualization technology's limitations, such as the number of supported CPUs and memory.

The public cloud flavor is called Windows Azure SQL Database (formerly called SQL Azure). It is hosted in Microsoft's data centers. Hardware, maintenance, HADR, and updates are all responsibilities of Microsoft. The customer is still responsible for index and query tuning, however.

Note Subsequent references to "Windows Azure SQL Database" will use the shorter form "SQL Database."

Using SQL Database, the customer can have multiple databases in the cloud server (a conceptual server, of course) but can connect to only one database at a time. The customer cannot switch between databases and cannot write multi-database queries.

The SQL Database engine is a specialized engine, although Microsoft uses the same code base as in the on-premises version. So the T-SQL features exposed in SQL Database are basically the same as those exposed locally. Most of the T-SQL that you will learn in this book is applicable to both on-premises and cloud flavors of SQL Server, but there are some exceptions, such as on-premises SQL

Server T-SQL features that are not yet implemented or exposed in SQL Database. Books Online for SQL Database details those features in the Transact-SQL Reference section at *http://msdn.microsoft.com/en-us/library/windowsazure/ee336281.aspx*. You should also note that the update and deployment rate of new versions of SQL Database is faster than that of an on-premises SQL Server. Therefore, it's possible that some T-SQL features may be exposed in SQL Database before they show up in an on-premises SQL Server version.

As mentioned, most of the T-SQL discussed in this book is either already available—or will be available—in SQL Database. The section in the Appendix that covers the installation of the sample database for this book also describes how to install the sample database in SQL Database, in case you already have access to it.

SQL Server Instances

A SQL Server instance, as illustrated in Figure 1-5, is an installation of a SQL Server database engine or service. You can install multiple instances of an on-premises SQL Server on the same computer. Each instance is completely independent of the others in terms of security, the data that it manages, and in all other respects. At the logical level, two different instances residing on the same computer have no more in common than two instances residing on two separate computers. Of course, same-computer instances do share the server's physical resources, such as CPU, memory, and disk.

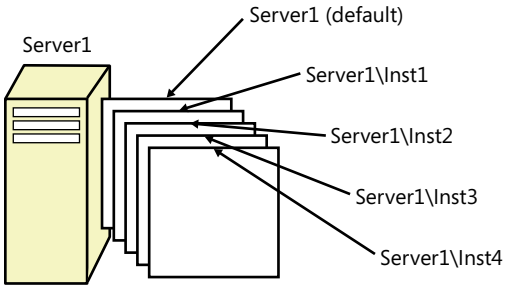

FIGURE 1-5 Multiple instances of SQL Server on the same computer.

You can set up one of the multiple instances on a computer as the *default instance*, whereas all others must be *named instances*. You determine whether an instance is the default or a named one upon installation; you cannot change that decision later. To connect to a default instance, a client application needs to specify the computer's name or IP address. To connect to a named instance, the client needs to specify the computer's name or IP address, followed by a backslash (\), followed by the instance name (as provided upon installation). For example, suppose you have two instances of SQL Server installed on a computer called *Server1*. One of these instances was installed as the default instance, and the other was installed as a named instance called *Inst1*. To connect to the default instance, you need to specify only *Server1* as the server name. However, to connect to the named instance, you need to specify both the server and the instance name: *Server1\Inst1*.

There are various reasons why you might want to install multiple instances of SQL Server on the same computer, but I'll mention only a couple here. One reason is to save on support costs. For example, to be able to test the functionality of features in response to support calls or reproduce errors that users encounter in the production environment, the support department needs local installations of SQL Server that mimic the user's production environment in terms of version, edition, and service pack of SQL Server. If an organization has multiple user environments, the support department needs multiple installations of SQL Server. Rather than having multiple computers, each hosting a different installation of SQL Server that must be supported separately, the support department can have one computer with multiple installed instances. Of course, you can achieve a similar result by using multiple virtual machines.

As another example, consider people like me who teach and lecture about SQL Server. For us, it is very convenient to be able to install multiple instances of SQL Server on the same laptop. This way, we can perform demonstrations against different versions of the product, showing differences in behavior between versions, and so on.

As a final example, providers of database services sometimes need to guarantee their customers complete security separation of their data from other customers' data. At least in the past, the database provider could have a very powerful data center hosting multiple instances of SQL Server, rather than needing to maintain multiple less-powerful computers, each hosting a different instance. More recently, cloud solutions and advanced virtualization technologies make it possible to achieve similar goals.

Databases

You can think of a database as a container of objects such as tables, views, stored procedures, and other objects. Each instance of SQL Server can contain multiple databases, as illustrated in Figure 1-6. When you install an on-premises flavor of SQL Server, the setup program creates several system databases that hold system data and serve internal purposes. After installation, you can create your own user databases that will hold application data.

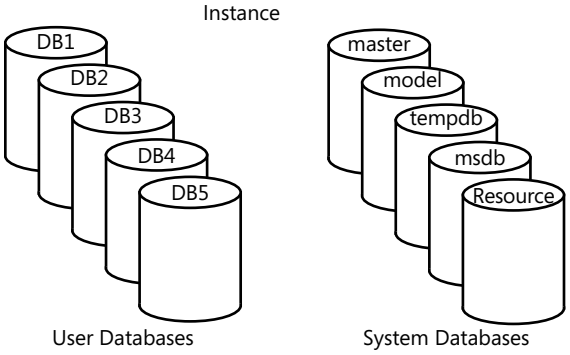

FIGURE 1-6 An example of multiple databases on a SQL Server instance.

The system databases that the setup program creates include *master*, *Resource*, *model*, *tempdb*, and *msdb*. A description of each follows.

- ***master*** The *master* database holds instance-wide metadata information, server configuration, information about all databases in the instance, and initialization information.
- ***Resource*** The *Resource* database is a hidden, read-only database that holds the definitions of all system objects. When you query system objects in a database, they appear to reside in the *sys schema* of the local database, but in actuality their definitions reside in the *Resource* database.
- ***model*** The *model* database is used as a template for new databases. Every new database that you create is initially created as a copy of *model*. So if you want certain objects (such as data types) to appear in all new databases that you create, or certain database properties to be configured in a certain way in all new databases, you need to create those objects and configure those properties in the *model* database. Note that changes you apply to the *model* database will not affect existing databases—only new databases that you create in the future.
- ***tempdb*** The *tempdb* database is where SQL Server stores temporary data such as work tables, sort space, row versioning information, and so on. SQL Server allows you to create temporary tables for your own use, and the physical location of those temporary tables is *tempdb*. Note that this database is destroyed and recreated as a copy of the *model* database every time you restart the instance of SQL Server.
- ***msdb*** The *msdb* database is where a service called SQL Server Agent stores its data. SQL Server Agent is in charge of automation, which includes entities such as jobs, schedules, and alerts. The SQL Server Agent is also the service in charge of replication. The *msdb* database also holds information related to other SQL Server features such as Database Mail, Service Broker, backups, and more.

In an on-premises installation of SQL Server, you can connect directly to the system databases *master*, *model*, *tempdb*, and *msdb*. In SQL Database, you can connect directly only to the system database *master*. If you create temporary tables or declare table variables (more on this topic in Chapter 10, "Programmable Objects"), they are created in *tempdb*, but you cannot connect directly to *tempdb* and explicitly create user objects there.

You can create multiple user databases within an instance (up to 32767). A user database holds objects and data for an application.

You can define a property called *collation* at the database level that will determine language support, case sensitivity, and sort order for character data in that database. If you do not specify a collation for the database when you create it, the new database will use the default collation of the instance (chosen upon installation).

To run T-SQL code against a database, a client application needs to connect to a SQL Server instance and be in the context of, or use, the relevant database.

In terms of security, to be able to connect to a SQL Server instance, the database administrator (DBA) must create a *logon* for you. In an on-premises SQL Server instance, the logon can be tied to your Windows credentials, in which case it is called a *Windows authenticated logon*. With a Windows authenticated logon, you won't need to provide logon and password information when connecting to SQL Server because you already provided those when you logged on to Windows. With both on-premises SQL Server and SQL Database, the logon can be independent of your Windows credentials, in which case it is called a *SQL Server authenticated logon*. When connecting to SQL Server using a SQL Server authenticated logon, you will need to provide both a logon name and a password.

The DBA needs to map your logon to a *database user* in each database that you are supposed to have access to. The database user is the entity that is granted permissions to objects in the database.

SQL Server 2012 supports a feature called *contained databases* that breaks the connection between a database user and a server-level logon. The user is fully contained within the specific database and is not tied to a logon at the server level. When creating the user, the DBA also provides a password. When connecting to SQL Server, the user needs to specify the database he or she is connecting to, as well as the user name and password, and the user cannot subsequently switch to other user databases.

So far, I've mainly mentioned the logical aspects of databases. If you're using SQL Database, your only concern is that logical layer. You do not deal with the physical layout of the database data and log files, *tempdb*, and so on. But if you're using on-premises SQL Server, you are responsible for the physical layer as well. Figure 1-7 shows a diagram of the physical database layout.

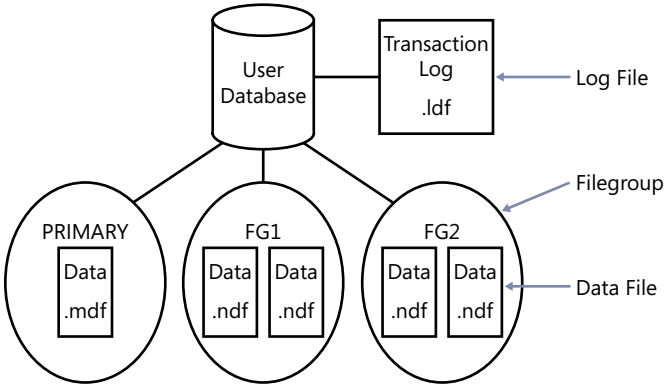

FIGURE 1-7 Database layout.

The database is made up of data files and transaction log files. When you create a database, you can define various properties for each file, including the file name, location, initial size, maximum size, and an autogrowth increment. Each database must have at least one data file and at least one log file (the default in SQL Server). The data files hold object data, and the log files hold information that SQL Server needs to maintain transactions.