Microsoft

T-SQL Fundamentals

Fourth Edition

Professional

Itzik Ben-Gan

Microsoft

T-SQL Fundamentals

Itzik Ben-Gan

T-SQL Fundamentals

Published with the authorization of Microsoft Corporation by:

Pearson Education, Inc.

ISBN-13: 978-0-13-810210-4

ISBN-10: 0-13-810210-4

Library of Congress Control Number: 2023930537

ScoutAutomatedPrintCode

Editor-in-Chief
Brett Bartow

Executive Editor
Loretta Yates

Associate Editor
Charvi Arora

Development Editor
Songlin Qiu

Managing Editor
Sandra Schroeder

Senior Project Editor
Tracey Croom

Copy Editor
Scout Festa

Indexer
Erika Milllen

Proofreader
Jen Hinchliffe

Technical Editor
Lilach Ben-Gan

Editorial Assistant
Cindy Teeters

Cover Designer
Twist Creative, Seattle

Compositor
codeMantra

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

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

—Thomas Campbell

Contents at a Glance

Contents

Acknowledgments

A number of people contributed to making this book a reality, either directly or indirectly, and deserve thanks and recognition. It's certainly possible I omitted some names unintentionally, and I apologize for this ahead of time.

To Lilach: You're the one who makes me want to be good at what I do. Besides being my inspiration in life, you always take an active role in my books, helping to review the text for the first time. In this book, you took a more official technical editing role, and I can't appreciate enough the errors you spotted, and the many ideas and suggestions for improvements.

To my siblings, Mickey and Ina: Thank you for the constant support and for accepting the fact that I'm away.

To Davide Mauri, Herbert Albert, Gianluca Hotz, and Dejan Sarka: Thanks for your valuable advice when I reached out asking for it.

To the editorial team at Pearson and related vendors. Loretta Yates, many thanks for being so good at what you do and for your positive attitude! Thanks to Charvi Arora for all your hard work and effort. Also, thanks to Songlin Qiu, Scout Festa, Karthik Orukaimani, and Tracey Croom for sifting through all the text and making sure it's polished.

To my friends from Lucient, Fernando G. Guerrero, Herbert Albert, Fritz Lechnitz, and many others. We've been working together for over two decades, and it's been quite a ride!

To members of the Microsoft SQL Server development team, Umachandar Jayachandran (UC), Conor Cunningham, Kevin Farlee, Craig Freedman, Kendal Van Dyke, Derek Wilson, Davide Mauri, Bob Ward, Buck Woody, and I'm sure many others. Thanks for creating such a great product, and thanks for all the time you spent meeting with me and responding to my emails, addressing my questions, and answering my requests for clarification.

To Aaron Bertrand, who besides being one of the most active and prolific SQL Server pros I know, does an amazing job editing the sqlperformance.com content, including my articles.

To Data Platform MVPs, past and present: Erland Sommarskog, Aaron Bertrand, Hugo Kornelis, Paul White, Alejandro Mesa, Tibor Karaszi, Simon Sabin, Denis Reznik, Tony Rogerson, and many others—and to the Data Platform MVP lead, Rie Merritt. This is a great program that I'm grateful for 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.

Finally, to my students: Teaching about T-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.

About the Author

ITZIK BEN-GAN is a leading authority on T-SQL, regularly teaching, lecturing, and writing on the subject. He has delivered numerous training events around the world focused on T-SQL Querying, Query Tuning, and Programming. He is the author of several books including *T-SQL Fundamentals*, *T-SQL Querying*, and *T-SQL Window Functions*. Itzik has been a Microsoft Data Platform MVP (Most Valuable Professional) since 1999.

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/IEC 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 a brief overview of programmable objects.

Although this book is intended for beginners, it's 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 might be considered advanced for readers who are new to T-SQL; therefore, you should consider those sections to be optional reading. If you feel comfortable with the material discussed in the book up to that point, you might want to tackle these more advanced subjects; otherwise, feel free to skip those sections and return to them after you gain more experience.

Many aspects of SQL are unique to the language and 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 relational terms 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 2022. When I discuss language elements that were introduced recently, I specify the version in which they were added.

Besides being available as an on-premises, or box, flavor, SQL Server is also available as cloud-based flavors called Azure SQL Database and Azure SQL Managed Instance. The code samples in this book are applicable to both the box and cloud flavors of SQL Server.

To complement the learning experience, the book provides exercises you can use to practice what you learn. I cannot emphasize enough the importance of working on those exercises, so make sure not to skip them!

Who Should Read This Book

This book is intended for T-SQL developers, database administrators (DBAs), business intelligence (BI) practitioners, data scientists, report writers, analysts, architects, and SQL Server power users who just started working with SQL Server and who need to write queries and develop code using T-SQL.

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

This book assumes that you are familiar with basic concepts of relational database management systems.

Organization of This Book

This book starts with a theoretical background to T-SQL querying and programming in Chapter 1, laying the foundation for the rest of the book, and provides basic coverage of creating tables and defining data integrity. The book covers various aspects of querying and modifying data in Chapters 2 through 8, and holds a discussion of transactions and concurrency in Chapter 10. In Chapter 9 and Chapter 11 the book covers specialized topics including temporal tables and SQL Graph. Finally, the book provides a brief overview of programmable objects in Chapter 12.

Here's a list of the chapters along with a short description of the content in each chapter:

- Chapter 1, "Background to T-SQL querying and programming," provides the theoretical background for SQL, set theory, and predicate logic. It examines relational theory, 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 (iTVFs), and the *APPLY* operator.

- Chapter 6, "Set operators," covers the set operators *UNION*, *INTERSECT*, and *EXCEPT*.
- Chapter 7, "T-SQL for data analysis," covers window functions, pivoting, unpivoting, working with grouping sets, and handling time-series data.
- Chapter 8, "Data modification," covers inserting, updating, deleting, and merging data.
- Chapter 9, "Temporal tables," covers system-versioned temporal tables.
- Chapter 10, "Transactions and concurrency," covers concurrency of user connections that work with the same data simultaneously; it covers transactions, locks, blocking, isolation levels, and deadlocks.
- Chapter 11, "SQL Graph," covers modeling data using graph-based concepts such as nodes and edges. It includes creating, modifying, and querying graph-based data.
- Chapter 12, "Programmable objects," provides a brief 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 *TSQLV6* sample database, start writing code against SQL Server, and learn how to get help by working with the product documentation.

System Requirements

The appendix, "Getting started," explains which editions of SQL Server 2022 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 described in the product documentation, under "Hardware and Software Requirements for Installing SQL Server 2022," at the following URL: *https://learn.microsoft.com/en-us/sql/sql-server/install/hardware-and-software-requirements-for-installing-sql-server-2022*. The appendix also explains how to work with the product documentation.

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

For the client tool to run the code samples against SQL Server, Azure SQL Database, and Azure SQL Managed Instance, you can use either SQL Server Management Studio (SSMS) or Azure Data Studio (ADS). You can download SSMS at *https://learn.microsoft.com/en-us/sql/ssms*. You can download Azure Data Studio at *https://learn.microsoft.com/en-us/sql/azure-data-studio*.

Code Samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All source code, including exercises and solutions, can be downloaded from the following webpage:

MicrosoftPressStore.com/TSQLFund4e/downloads

Follow the instructions to download the TSQLFundamentalsYYYYMMDD.zip file, where *YYYYMMDD* reflects the last update date of the source code.

Refer to the appendix, "Getting started," for details about the source code.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

MicrosoftPressStore.com/TSQLFund4e/errata

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit *MicrosoftPressStore.com/Support*

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

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. T-SQL is more than just a language—it's a way of thinking. In my first few books about T-SQL, I've written extensively on advanced topics, and for years I have postponed writing about fundamentals. This is not because T-SQL fundamentals are simple or easy—in fact, it's 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 its 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 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 an extension to, SQL in SQL Server—its on-premises RDBMS flavor, and in Azure SQL and Azure Synapse Analytics—its cloud-based RDBMS flavors.

Note The term *Azure SQL* collectively refers to three different cloud offerings: Azure SQL Database, Azure SQL Managed Instance, and SQL Server on Azure VM. I describe the differences between these offerings later in the chapter.

This section provides a brief theoretical background about SQL, set theory and predicate logic, the relational model, and types of database systems. 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 apply data management and manipulation following the relational model's principles with languages other than SQL—for example, with C# in an object model. Today it is common to see RDBMSs that support languages other than just a dialect of SQL—for example, the integration of the CLR, Java, Python, and R in SQL Server, with which you can handle tasks that historically you handled mainly with SQL, such as data manipulation.

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 de facto language used by virtually all leading RDBMSs.

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, 3rd Edition,* by C. J. Date (O'Reilly Media, 2015).

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 its 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), SQL:2011 (2011), and SQL:2016 (2016). The SQL standard is made of multiple parts. Part 1 provides the framework and Part 2 defines the foundation with the core SQL elements. The other parts define standard extensions, such as SQL for XML, SQL-Java integration, and others.

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. Moreover, T-SQL does not implement all of standard SQL. When describing a language element for the first time, I'll typically mention if it's nonstandard.

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.*
>
> —Georg Cantor: His Mathematics and Philosophy of the Infinite, by *Joseph W. Dauben (Princeton University Press, 2020)*

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 (*columns* in SQL) that make up the heading of a relation (*table* in SQL), an element (in this case, an attribute) is supposed to be identified by name—not by ordinal position.

Similarly, consider the set of tuples (*rows* in SQL) that make up the body of the relation; an element (in this case a tuple) 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 ordered in a specific way, perhaps for presentation purposes.

Predicate logic

Predicate logic, whose roots go 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 the 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 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 nonmathematicians, these concepts can be quite intimidating. The sections that follow cover some 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*). Note that operations on relations (based on relational algebra) result in a relation (for example, an intersection between two relations). This is what's known as the *closure* property of the relational algebra, and is what enables the nesting of relational expressions.

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, as Figure 1-1 shows, a relation is made of a heading and a body. The heading consists of a set of attributes (*columns* in SQL), where each element has a name and a type name and is identified by name. The body consists of a set of tuples (*rows* in SQL), where each element is identified by a key. To keep things simple, I'll often refer to a table as a set of rows.

Figure 1-1 shows an illustration of a relation called Employees. It compares the components of a relation in relational theory with those of a table in SQL.

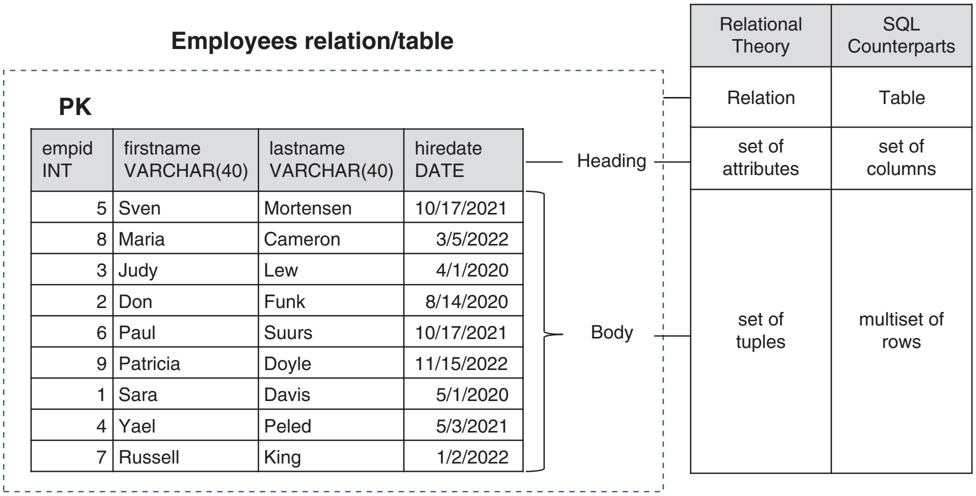

FIGURE 1-1 Illustration of Employees relation

Be aware that creating a truly adequate visual representation of a relation is very difficult in practice, since the set of attributes making the heading of a relation has no order, and the same goes for the set of tuples making the body of a relation. In an illustration, it might seem like those elements do have order even though they don't. Just make sure to keep this in mind.

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 Jiru Ben-Gan was born on June 22, 2003, and works in the Pet Food 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 *closed-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 has a name and a type name, and is identified by 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, and *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, 2003 (not to mention a birth date stated as something like "abc!"). Note that types are not restricted to base types such as integers or dates; a type can also be an enumeration of possible values, such as an enumeration of possible job positions. A type can be simple or 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 is a geometry type that supports polygons.

Missing values

There's an aspect of the relational model and SQL that is the source of many passionate debates. Whether to support the notion of missing values and three-valued predicate 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 support the idea of three-valued predicate logic, taking into account cases where values are missing. A predicate involving a missing value yields neither *true* nor *false* as the result truth value—it yields *unknown*.

Take, for example, a *mobilephone* 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? One option is to have the *mobilephone* attribute allow the use of a special marker for a missing value. Then a predicate used for filtering purposes, comparing the *mobilephone* attribute with some specific number, will yield *unknown* for the case with the missing value. Three-valued predicate logic refers to the three possible truth values that can result from a predicate—*true*, *false*, and *unknown*.

Some people believe that *NULLs* and three-valued predicate logic are nonrelational, whereas others believe that they are relational. Codd actually advocated for four-valued predicate logic, saying that there were two different cases of missing values: missing but applicable (A-Values marker), and missing but inapplicable (I-Values marker). 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 doesn't make a distinction between the two cases for missing values that Codd does; rather, it defines the *NULL* marker to signify any kind of missing value. It also supports three-valued predicate logic. Support for *NULLs* 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 and representing missing values with your own custom means—is not necessarily less problematic.

> **Note** As mentioned, a *NULL* is not a value but rather a marker for a missing value. Therefore, though unfortunately it's common, the use of the terminology "*NULL* value" is incorrect. The correct terminology is "*NULL* marker" or just "*NULL*." In the book, I typically use the latter because it's more common in the SQL community.

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 and "nullability" (whether it supports or doesn't support *NULLs*). 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 the enforcement of *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 of a relation. Based on a candidate key's attribute values you can uniquely identify a tuple (row). A constraint enforcing a candidate key prevents duplicates. You can identify multiple candidate keys in a relation. For example, in an *Employees* relation, you can have one candidate key based on *employeeid*, another 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. You can find the definition of ERM in the following Wikipedia article: *https://en.wikipedia.org/wiki/Entity–relationship_model*.

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 enforce the uniqueness of rows in SQL by defining a primary key or unique constraint in the table.

You can operate on attributes only 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 attributes (*firstname* and *lastname*), or three attributes (*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 applications that use it, an attribute might also be subatomic. For example, if an address attribute is considered atomic for the applications that use it, 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*, *qty2020*, *qty2021*, and *qty2022*. 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: 2020, 2021, and 2022.

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 nonkey and candidate-key attributes. For every candidate key, every nonkey attribute has to be fully functionally dependent on the entire candidate key. In other words, a nonkey attribute cannot be fully functionally dependent on part of a candidate key. To put it more informally, if you need to obtain any nonkey 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 1-2.) The *Orders* relation contains the following attributes: *orderid*, *productid*, *orderdate*, *qty*, *customerid*, and *companyname*. The primary key is defined on *orderid* and *productid*.

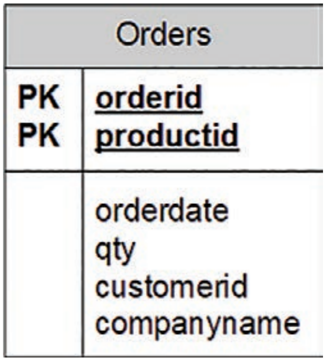

FIGURE 1-2 Data model before applying 2NF

The second normal form is violated in Figure 1-2 because there are nonkey 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 1-3). 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*.

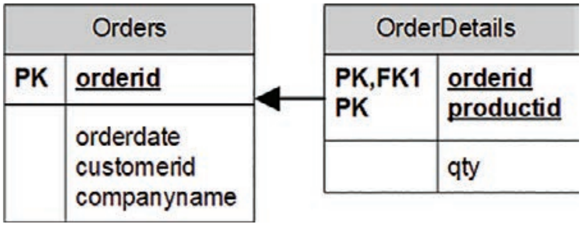

FIGURE 1-3 Data model after applying 2NF and before 3NF

3NF

The third normal form also has two rules. The data must meet the second normal form. Also, all nonkey attributes must be dependent on candidate keys nontransitively. Informally, this rule means that all nonkey attributes must be mutually independent. In other words, one nonkey attribute cannot be dependent on another nonkey 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 1-4) with the attributes *customerid* (as the primary key) and *companyname*. Then you can remove the *companyname* attribute from the *Orders* relation.

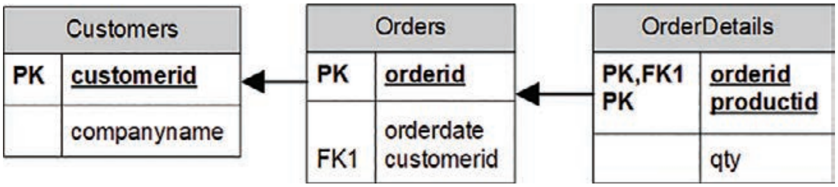

FIGURE 1-4 Data model after applying 3NF

Informally, 2NF and 3NF are commonly summarized as follows: "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.

> **Note** SQL, as well as T-SQL, permits violating all the normal forms in real tables. It's the data modeler's prerogative and responsibility to design a normalized model.

Types of database workloads

Two main types of workloads use Microsoft RDBMS platforms and T-SQL to manage and manipulate the data: online transactional processing (OLTP) and data warehouses (DWs). The former can be implemented on SQL Server or Azure SQL. The latter can be implemented on SQL Server or Azure SQL, which use a symmetric multiprocessing (SMP) architecture; or, for more demanding workloads, on Synapse Azure Analytics, which uses a massively parallel processing (MPP) architecture. Figure 1-5 illustrates those workloads and systems and the transformation process that usually takes place between them.

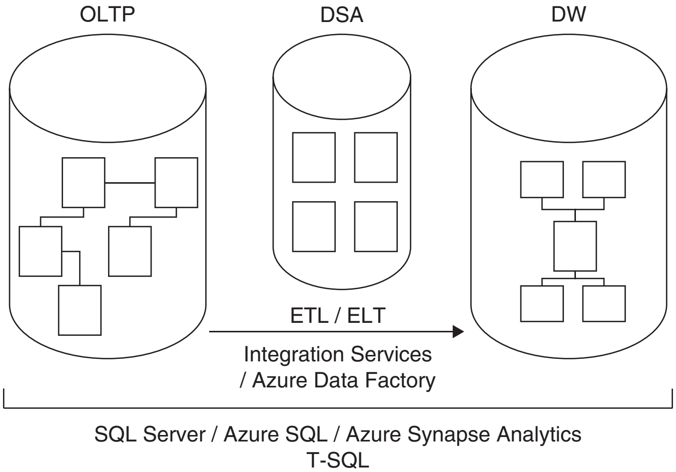

FIGURE 1-5 Classes of database systems

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

- OLTP: online transactional processing
- DSA: data-staging area
- DW: data warehouse
- ETL / ELT: extract, transform, and load; or extract, load, and transform

Online transactional processing

Data is entered initially into an online transactional processing system. The primary 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 or Azure SQL 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 than 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 dimensions such as customers, products, employees, and time.

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*. A star schema is considered a special case of a snowflake 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 in SQL Server or Azure SQL, which use a SMP architecture. You can implement a more demanding workload on Azure Synapse Analytics, which uses a MPP architecture. In any case, you query and manage the data warehouse 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*. Some of the integration solutions—especially cloud-based—extract the data, load it, and then transform it. In such a case, the process is known by the acronym *ELT*. Microsoft provides an on-premises tool called Microsoft SQL Server Integration Services (SSIS) to handle ETL/ELT needs, which comes with a SQL Server license. Microsoft also provides a serverless cloud service for ETL/ELT solutions called Azure Data Factory.

Often the ETL/ELT process will involve the use of a data-staging area (DSA) between the OLTP and the DW. The DSA can reside in a relational database, such as SQL Server or Azure SQL, or in Azure Data Lake Storage Gen2, and is used as the data-cleansing area.

SQL Server architecture

This section will introduce you to the SQL Server architecture, the on-premises and cloud RDBMS flavors that Microsoft offers, the entities involved—SQL Server instances, databases, schemas, and database objects—and the purpose of each entity.

On-premises and cloud RDBMS flavors

Initially, Microsoft offered mainly one enterprise-level RDBMS—an on-premises flavor called Microsoft SQL Server. These days, Microsoft offers an overwhelming plethora of options as part of its data platform, which constantly keeps evolving. Within its data platform, Microsoft offers both on-premises, or box, solutions, and service-based cloud solutions.

On-premises

The on-premises RDBMS flavor that Microsoft offers is called Microsoft SQL Server, or just SQL Server. This is the traditional flavor, usually installed on the customer's premises. The customer is responsible for everything—getting the hardware, installing the software, patching, high availability and disaster recovery, security, and everything else.

The customer can install multiple instances of the product on 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 (defined later).

The querying language used is T-SQL. You can run all 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 SQL Server, as well as creating the sample database.

Cloud

Cloud computing provides compute and storage resources on demand from a shared pool of resources. Microsoft's RDBMS technologies can be provided both as private-cloud and public-cloud services. A *private cloud* is cloud infrastructure that services a single organization and usually uses virtualization technology. It's typically hosted locally at the customer site, and maintained by the IT group in the organization. It's about self-service agility, allowing the users to deploy resources on demand. It provides standardization and usage metering. The database engine is usually an on-premises engine, where T-SQL is used to manage and manipulate the data. SQL Server can run on either Windows or Linux, and therefore can be deployed on any private cloud, no matter the underlying OS platform.

As for the public cloud, the services are provided over the network and available to the public. Microsoft provides two forms of public RDBMS cloud services: infrastructure as a service (IaaS) and platform as a service (PaaS). With IaaS, you provision a virtual machine (VM) that resides in Microsoft's cloud infrastructure. This offering is known as SQL Server on Azure VM. As a starting point, you can choose between several preconfigured VMs that already have a certain version and edition of SQL Server installed on them, and follow best practices. The hardware is maintained by Microsoft, but you're responsible for maintaining and patching the software. It's essentially like maintaining your own SQL Server installation—one that happens to reside on Microsoft's hardware.

With PaaS, Microsoft provides the database cloud platform as a service. It's hosted in Microsoft's data centers. Hardware, software installation and maintenance, high availability and disaster recovery, and patching are all responsibilities of Microsoft. The customer is still responsible for index and query tuning, however.

Microsoft provides a number of PaaS database offerings. For OLTP systems as well as SMP-based data warehouses, it offers Azure SQL Database and Azure SQL Managed Instance. You can find a detailed comparison between these two PaaS offerings here: *https://learn.microsoft.com/en-us/azure/azure-sql/database/features-comparison*. You will find that, for example, with the former you cannot perform cross-database/three-part name queries and with the latter you can, as well as other differences. Generally, the latter gives you closer parity with the on-premises flavor.

As mentioned, Microsoft uses the term *Azure SQL* to collectively refer to the three SMP-based cloud offerings: SQL Server on Azure VM, Azure SQL Database, and Azure SQL Managed Instance.

Note that Azure SQL Database and Azure SQL Managed Instance share the same code base with the latest version of SQL Server. So most of the T-SQL language surface is the same in both the on-premises and cloud environments. Therefore, most of the T-SQL you'll learn about in this book is applicable to both environments. You can read about the differences that do exist—especially between SQL Server and Azure SQL Database—here: *https://learn.microsoft.com/en-us/azure/azure-sql/database/transact-sql-tsql-differences-sql-server*. You should also note that the update and deployment rate of the cloud flavors are faster than that of the on-premises SQL Server product. Therefore, some T-SQL features might be exposed in the cloud first before they show up in the on-premises product.

As mentioned, Microsoft also provides a MPP-based PaaS offering called Azure Synapse Analytics as a cloud native data warehousing solution, with a distributed processing engine, that you query and manage with T-SQL.

SQL Server instances

In the on-premises product, an instance of SQL Server, as illustrated in Figure 1-6, is an installation of a SQL Server database engine or service. You can install multiple instances of SQL Server on the same computer. Each instance is completely independent of the others in terms of security and 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.

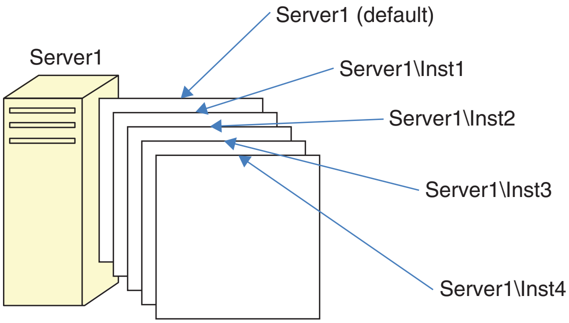

FIGURE 1-6 Multiple instances of SQL Server on the same computer

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. 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 a couple of them here. One reason, mainly historic, is to save on support costs. For example, 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, nowadays you can meet the same needs with container or virtualization technologies. It's just that SQL Server instances were available before virtualization and container technologies took off.

As another example, consider people like me who teach and lecture about SQL Server. For us, it is 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, also mainly historic, 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. Nowadays, cloud solutions and advanced container and 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 1-7. When you install SQL Server, the setup program creates several system databases that hold system data and serve internal purposes. After you install SQL Server, you can create your own user databases that will hold application data.

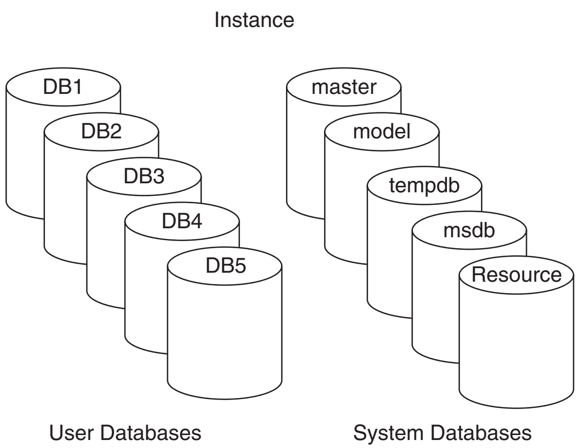

FIGURE 1-7 An example of multiple databases on a SQL Server instance

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

- ***master*** The *master* database holds instance-wide metadata information, the server configuration, information about all databases in the instance, and initialization information.
- ***model*** The *model* database is used as a template for new databases. Every new database you create is initially created as a copy of *model*. So if you want certain objects (such as user-defined

data types) to appear in all new databases 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 you create in the future.

- ***tempdb*** The *tempdb* database is where SQL Server stores temporary data such as work tables, sort and hash table data when it needs to persist those, row versioning information, and so on. SQL Server allows you to create temporary tables for your own use, and the physical location of those is in *tempdb*. Note that this database is destroyed and re-created as a copy of the *model* database every time you restart the instance of SQL Server.
- ***msdb*** The *msdb* database is used mainly by a service called SQL Server Agent to store its data. SQL Server Agent is in charge of automation, which includes entities such as jobs, schedules, and alerts. 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.
- ***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.

In SQL Server and Azure SQL Managed Instance, you can connect directly to the system databases *master*, *model*, *tempdb*, and *msdb*. In Azure 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 12, "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 (up to 32,767) within an instance. A user database holds objects and data for an application.

You can define a property called *collation* at the database level that will determine default 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. The application can still access objects from other databases by adding the database name as a prefix. That's the case with both SQL Server and Azure SQL Managed Instance. Azure SQL Database does not support cross-database/three-part name queries.

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

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

SQL Server supports a feature called *contained databases* that breaks the connection between a database user and an instance-level login. The user (Windows or SQL authenticated) is fully contained within the specific database and is not tied to a login at the instance level. When connecting to SQL Server, the user needs to specify the database he or she is connecting to, 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 Azure SQL Database or Azure SQL Managed Instance, your only concern is that logical layer. You do not deal with the physical layout of the database's data and log files, *tempdb*, and so on. But if you're using SQL Server (including SQL Server on Azure VM), you are responsible for the physical layer as well. Figure 1-8 shows a diagram of the physical database layout.

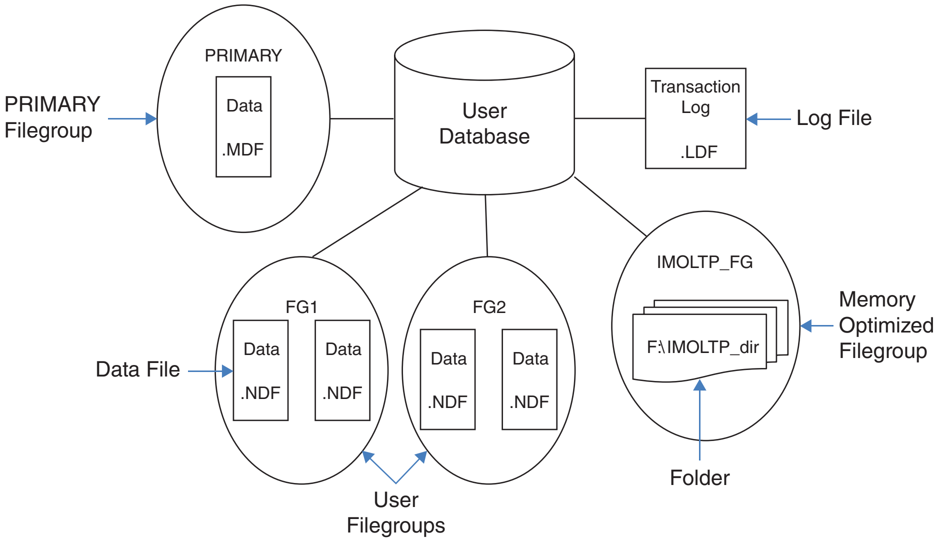

FIGURE 1-8 Database layout

The database is made up of data files, transaction log files, and optionally checkpoint files holding memory-optimized data (part of a feature called *In-Memory OLTP*, which I describe shortly). When you create a database, you can define various properties for data and log files, 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.

Although SQL Server can write to multiple data files in parallel, it can write to only one log file at a time, in a sequential manner. Therefore, unlike with data files, having multiple log files does not result in a performance benefit. You might need to add log files if the disk drive where the log resides runs out of space.

Data files are organized in logical groups called *filegroups*. A filegroup is the target for creating an object, such as a table or an index. The object data will be spread across the files that belong to the target filegroup. Filegroups are your way of controlling the physical locations of your objects. A database must have at least one filegroup called *PRIMARY*, and it can optionally have other user filegroups as well. The *PRIMARY* filegroup contains the primary data file (which has an .mdf extension) for the database, and the database's system catalog. You can optionally add secondary data files (which have an .ndf extension) to *PRIMARY*. User filegroups contain only secondary data files. You can decide which filegroup is marked as the default filegroup. Objects are created in the default filegroup when the object creation statement does not explicitly specify a different target filegroup.

File extensions .mdf, .ldf, and .ndf

The database file extensions .mdf and .ldf are straightforward. The extension *.mdf* stands for *Master Data File* (not to be confused with the *master* database), and *.ldf* stands for *Log Data File*. According to one anecdote, when discussing the extension for the secondary data files, one of the developers suggested, humorously, using .ndf to represent "Not Master Data File," and the idea was accepted.

The SQL Server database engine includes a memory-optimized engine called In-Memory OLTP. You can use this feature to integrate memory-optimized objects, such as memory-optimized tables and natively compiled modules (procedures, functions, and triggers), into your database. To do so, you need to create a filegroup in the database marked as containing memory-optimized data and, within it, at least one path to a folder. SQL Server stores checkpoint files with memory-optimized data in that folder, and it uses those to recover the data every time SQL Server is restarted.

Schemas and objects

When I said earlier that a database is a container of objects, I simplified things a bit. As illustrated in Figure 1-9, a database contains schemas, and schemas contain objects. You can think of a schema as a container of objects, such as tables, views, stored procedures, and others.

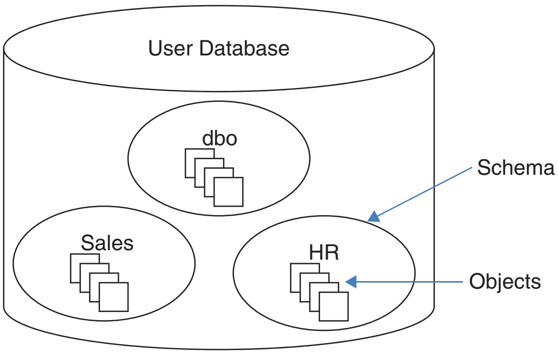

FIGURE 1-9 A database, schemas, and database objects

You can control permissions at the schema level. For example, you can grant a user SELECT permissions on a schema, allowing the user to query data from all objects in that schema. So security is one of the considerations for determining how to arrange objects in schemas.

The schema is also a namespace—it is used as a prefix to the object name. For example, suppose you have a table named *Orders* in a schema named *Sales*. The schema-qualified object name (also known as the *two-part object name*) is *Sales.Orders*. You can refer to objects in other databases by adding the database name as a prefix (*three-part object name*), and to objects in other instances by adding the instance name as a prefix (*four-part object name*). If you omit the schema name when referring to an object, SQL Server will apply a process to resolve the schema name, such as checking whether the object exists in the user's default schema and, if the object doesn't exist, checking whether it exists in the *dbo* schema. Microsoft recommends that when you refer to objects in your code you always use the two-part object names. If multiple objects with the same name exist in different schemas, you might end up getting a different object than the one you wanted.

Creating tables and defining data integrity

This section describes the fundamentals of creating tables and defining data integrity using T-SQL. Feel free to run the included code samples in your environment.

> **More Info**
>
> If you don't know yet how to run code against SQL Server, the Appendix will help you get started.

As mentioned earlier, DML rather than DDL is the focus of this book. Still, you need to understand how to create tables and define data integrity. I won't go into great detail here, but I'll provide a brief description of the essentials.

Before you look at the code for creating a table, remember that tables reside within schemas, and schemas reside within databases. The examples use the book's sample database, *TSQLV6*, and a schema called *dbo*.

> **More Info**
>
> See the Appendix for details on creating the sample database.

The examples here use a schema named *dbo* that is created automatically in every database and is also used as the default schema for users who are not explicitly associated with a different schema.

Creating tables

The following code creates a table named *Employees* in the *dbo* schema in the *TSQLV6* database:

```
USE TSQLV6;

DROP TABLE IF EXISTS dbo.Employees;

CREATE TABLE dbo.Employees
(
  empid     INT         NOT NULL,
  firstname VARCHAR(30) NOT NULL,
  lastname  VARCHAR(30) NOT NULL,
  hiredate  DATE        NOT NULL,
  mgrid     INT         NULL,
  ssn       VARCHAR(20) NOT NULL,
  salary    MONEY       NOT NULL
);
```

The *USE* statement sets the current database context to that of *TSQLV6*. It is important to incorporate the *USE* statement in scripts that create objects to ensure that SQL Server creates the objects in the specified database. In SQL Server and Azure SQL Managed Instance, the *USE* statement can actually change the database context from one to another. In Azure SQL Database, you cannot switch between different databases, but the *USE* statement will not fail as long as you are already connected to the target database. So even in Azure SQL Database, I recommend having the *USE* statement to ensure that you are connected to the right database when creating your objects.

The *DROP TABLE IF EXISTS* statement (aka DIE) drops the *Employees* table if it already exists in the current database. Of course, you can choose a different treatment, such as not creating the object if it already exists.

You use the *CREATE TABLE* statement to define a table. You specify the name of the table and, in parentheses, the definition of its attributes (columns).

Notice the use of the two-part name *dbo.Employees* for the table name, as recommended earlier. If you omit the schema name, for ad-hoc queries SQL Server will assume the default schema associated with the database user running the code. For queries in stored procedures, SQL Server will assume the schema associated with the procedure's owner.

For each attribute, you specify the attribute name, data type, and whether the value can be *NULL* (which is called *nullability*).

In the *Employees* table, the attributes *empid* (employee ID) and *mgrid* (manager ID) are each defined with the *INT* (four-byte integer) data type; the *firstname*, *lastname*, and *ssn* (US Social Security number) are defined as *VARCHAR* (variable-length character string with the specified maximum supported number of characters); *hiredate* is defined as *DATE* and *salary* is defined as *MONEY*.

If you don't explicitly specify whether a column allows or disallows *NULLs*, SQL Server will have to rely on defaults. Standard SQL dictates that when a column's nullability is not specified, the assumption should be *NULL* (allowing *NULLs*), but SQL Server has settings that can change that behavior. I recommend that you be explicit and not rely on defaults. Also, I recommend defining a column as *NOT NULL* unless you have a compelling reason to support *NULLs*. If a column is not supposed to allow *NULLs* and you don't enforce this with a *NOT NULL* constraint, you can rest assured that *NULLs* will occur. In the *Employees* table, all columns are defined as *NOT NULL* except for the *mgrid* column. A *NULL* in the *mgrid* column would represent the fact that the employee has no manager, as in the case of the CEO of the organization.

Coding style

You should be aware of a few general notes regarding coding style, the use of white spaces (space, tab, new line, and so on), and semicolons. My advice is that you use a style that you and your fellow developers feel comfortable with. What ultimately matters most is the consistency, readability, and maintainability of your code. I have tried to reflect these aspects in my code throughout the book.

T-SQL lets you use white spaces quite freely in your code. You can take advantage of white spaces to facilitate readability. For example, I could have written the code in the previous section as a single line. However, the code wouldn't have been as readable as when it is broken into multiple lines that use indentation.

The practice of using a semicolon to terminate statements is standard and, in fact, is a requirement in several other database platforms. T-SQL requires the semicolon only in particular cases—but in cases where it's not required, it's still allowed. I recommend that you adopt the practice of terminating all statements with a semicolon. Not only will doing this improve the readability of your code, but in some cases it can save you some grief. (When a semicolon is required and is *not* specified, the error message SQL Server produces is not always clear.)

Note The SQL Server documentation indicates that not terminating T-SQL statements with a semicolon is a deprecated feature. That's one more reason to get into the habit of terminating all your statements with a semicolon, even where it's currently not required.

Defining data integrity

As mentioned earlier, one of the great benefits of the relational model is that data integrity is an integral part of it. Data integrity enforced as part of the model—namely, as part of the table definitions—is considered *declarative data integrity*. Data integrity enforced with code—such as with stored procedures or triggers—is considered *procedural data integrity*.

Data type and nullability choices for attributes and even the data model itself are examples of declarative data integrity constraints. In this section, I describe other examples of declarative constraints: primary key, unique, foreign key, check, and default constraints. You can define such constraints when creating a table as part of the *CREATE TABLE* statement, or you can define them for already created tables by using an *ALTER TABLE* statement. All types of constraints except for default constraints can be defined as *composite constraints*—that is, based on more than one attribute.

Primary key constraints

A primary key constraint enforces the uniqueness of rows and also disallows *NULLs* in the constraint attributes. Each unique combination of values in the constraint attributes can appear only once in the table—in other words, only in one row. An attempt to define a primary key constraint on a column that allows *NULLs* will be rejected by the RDBMS. Each table can have only one primary key.

Here's an example of defining a primary key constraint on the *empid* attribute in the *Employees* table that you created earlier:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT PK_Employees
  PRIMARY KEY(empid);
```

With this primary key in place, you can be assured that all *empid* values will be unique and known. An attempt to insert or update a row such that the constraint would be violated will be rejected by the RDBMS and result in an error.

To enforce the uniqueness of the logical primary key constraint, SQL Server will create a unique index behind the scenes. A *unique index* is a physical object used by SQL Server to enforce uniqueness. Indexes (not necessarily unique ones) are also used to speed up queries by avoiding sorting and unnecessary full table scans (similar to indexes in books).

Unique constraints

A unique constraint enforces the uniqueness of rows, allowing you to implement the concept of alternate keys from the relational model in your database. Unlike with primary keys, you can define multiple unique constraints within the same table. Also, a unique constraint is not restricted to columns defined as *NOT NULL*. More on the specifics of *NULL*-handling shortly.

The following code defines a unique constraint on the *ssn* column in the *Employees* table:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT UNQ_Employees_ssn
  UNIQUE(ssn);
```

As with a primary key constraint, SQL Server will create a unique index behind the scenes as the physical mechanism to enforce the logical unique constraint.

For the purpose of enforcing a unique constraint, SQL Server handles *NULLs* just like non-*NULL* values. Consequently, for example, a single-column unique constraint allows only one *NULL* in the constrained column. However, the SQL standard defines *NULL*-handling by a unique constraint differently, like so: *"A unique constraint on T is satisfied if and only if there do not exist two rows R1 and R2 of T such that R1 and R2 have the same non-NULL values in the unique columns."* In other words, only the non-*NULL* values are compared to determine whether duplicates exist. Consequently, a standard single-column unique constraint would allow multiple *NULLs* in the constrained column. To emulate a standard single-column unique constraint in SQL Server you can use a unique filtered index that filters only non-*NULL* values. For example, suppose that the column *ssn* allowed *NULLs*, and you wanted to create such an index instead of a unique constraint. You would have used the following code:

```
CREATE UNIQUE INDEX idx_ssn_notnull ON dbo.Employees(ssn) WHERE ssn IS NOT NULL;
```

The index is defined as a unique one, and the filter excludes *NULLs* from the index, so duplicate *NULLs* will be allowed in the underlying table, whereas duplicate non-*NULL* values won't be allowed.

Emulating a standard composite unique constraint in SQL Server is a bit more involved and may not be of common knowledge. You can find the details in the following article: *https://sqlperformance.com/2020/03/t-sql-queries/null-complexities-part-4-missing-standard-unique-constraint*.

Foreign key constraints

A foreign key enforces referential integrity. This constraint is defined on one or more attributes in what's called the *referencing* table and points to candidate key (primary key or unique constraint) attributes in what's called the *referenced* table. Note that the referencing and referenced tables can be one and the same. The foreign key's purpose is to restrict the values allowed in the foreign key columns to those that exist in the referenced columns.

The following code creates a table called *Orders* with a primary key defined on the *orderid* column:

```
DROP TABLE IF EXISTS dbo.Orders;

CREATE TABLE dbo.Orders
(
  orderid   INT         NOT NULL,
  empid     INT         NOT NULL,
  custid    VARCHAR(10) NOT NULL,
  orderts   DATETIME2   NOT NULL,
  qty       INT         NOT NULL,
  CONSTRAINT PK_Orders
    PRIMARY KEY(orderid)
);
```

Suppose you want to enforce an integrity rule that restricts the values supported by the *empid* column in the *Orders* table to the values that exist in the *empid* column in the *Employees* table. You can achieve this by defining a foreign key constraint on the *empid* column in the *Orders* table pointing to the *empid* column in the *Employees* table, like so:

```
ALTER TABLE dbo.Orders
  ADD CONSTRAINT FK_Orders_Employees
  FOREIGN KEY(empid)
  REFERENCES dbo.Employees(empid);
```

Similarly, if you want to restrict the values supported by the *mgrid* column in the *Employees* table to the values that exist in the *empid* column of the same table, you can do so by adding the following foreign key:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT FK_Employees_Employees
  FOREIGN KEY(mgrid)
  REFERENCES dbo.Employees(empid);
```

Note that *NULLs* are allowed in the foreign key columns (*mgrid* in the last example) even if there are no *NULLs* in the referenced candidate key columns.

The preceding two examples are basic definitions of foreign keys that enforce a referential action called *no action*. No action means that attempts to delete rows from the referenced table or update the referenced candidate key attributes will be rejected if related rows exist in the referencing table. For example, if you try to delete an employee row from the *Employees* table when there are related orders in the *Orders* table, the RDBMS will reject such an attempt and produce an error.

You can define the foreign key with actions that will compensate for such attempts (to delete rows from the referenced table or update the referenced candidate key attributes when related rows exist in the referencing table). You can define the options *ON DELETE* and *ON UPDATE* with actions such as *CASCADE*, *SET DEFAULT*, and *SET NULL* as part of the foreign key definition. *CASCADE* means that the operation (delete or update) will be cascaded to related rows. For example, *ON DELETE CASCADE* means that when you delete a row from the referenced table, the RDBMS will delete the related rows from the referencing table. *SET DEFAULT* and *SET NULL* mean that the compensating action will set the foreign key attributes of the related rows to the column's default value or *NULL*, respectively. Note that regardless of which action you choose, the referencing table will have orphaned rows only in the case of the exception with *NULLs* in the referencing column that I mentioned earlier. Parent rows with no related child rows are always allowed.

Check constraints

You can use a *check constraint* to define a predicate that a row must meet to be entered into the table or to be modified. For example, the following check constraint ensures that the salary column in the *Employees* table will support only positive values:

```
ALTER TABLE dbo.Employees
  ADD CONSTRAINT CHK_Employees_salary
  CHECK(salary > 0.00);
```

An attempt to insert or update a row with a nonpositive salary value will be rejected by the RDBMS. Note that a check constraint rejects an attempt to insert or update a row when the predicate evaluates to *FALSE*. The modification will be accepted when the predicate evaluates to either *TRUE* or *UNKNOWN*. For example, salary –1000 will be rejected, whereas salaries 50000 and *NULL* will both be accepted (if the column allowed *NULLs*). As mentioned earlier, SQL is based on three-valued logic, which results in two actual actions. With a check constraint, the row is either accepted or rejected.

When adding check and foreign key constraints, you can specify an option called *WITH NOCHECK* that tells the RDBMS you want it to bypass constraint checking for existing data. This is considered a bad practice because you cannot be sure your data is consistent. You can also disable or enable existing check and foreign key constraints.

Default constraints

A default constraint is associated with a particular attribute. It's an expression that is used as the default value when an explicit value is not specified for the attribute when you insert a row. For example, the following code defines a default constraint for the *orderts* attribute (representing the order's time stamp):

```
ALTER TABLE dbo.Orders
  ADD CONSTRAINT DFT_Orders_orderts
  DEFAULT(SYSDATETIME()) FOR orderts;
```

The default expression invokes the *SYSDATETIME* function, which returns the current date and time value. After this default expression is defined, whenever you insert a row into the *Orders* table and do not explicitly specify a value in the *orderts* attribute, SQL Server will set the attribute value to *SYSDATETIME*.

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS dbo.Orders, dbo.Employees;
```

Conclusion

This chapter provided a brief background to T-SQL querying and programming. It presented a theoretical background, explaining the strong foundations that T-SQL is based on. It gave an overview of the SQL Server architecture and concluded with sections that demonstrated how to use T-SQL to create tables and define data integrity. I hope that by now you see that there's something special about SQL, and that it's not just a language that can be learned as an afterthought. This chapter equipped you with fundamental concepts—the actual journey is just about to begin.

Index

Symbols and Numbers

A

B

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y-Z