

T-SQL Fundamentals

Fourth Edition



Professional



Itzik Ben-Gan

FREE SAMPLE CHAPTER |



T-SQL Fundamentals

Itzik Ben-Gan

T-SQL Fundamentals

Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

Copyright © 2023 by Itzik Ben-Gan.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

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

ISBN-10: 0-13-810210-4

Library of Congress Control Number: 2023930537

ScoutAutomatedPrintCode

Trademarks

Microsoft and the trademarks listed at <http://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author, the publisher, and Microsoft Corporation shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs accompanying it.

Special Sales

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

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

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

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 Millen

Proofreader

Jan 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

	<i>Acknowledgments</i>	<i>xxi</i>
	<i>About the Author</i>	<i>xxiii</i>
	<i>Introduction</i>	<i>xxiv</i>
CHAPTER 1	Background to T-SQL querying and programming	1
CHAPTER 2	Single-table queries	27
CHAPTER 3	Joins	117
CHAPTER 4	Subqueries	149
CHAPTER 5	Table expressions	177
CHAPTER 6	Set operators	211
CHAPTER 7	T-SQL for data analysis	231
CHAPTER 8	Data modification	293
CHAPTER 9	Temporal tables	343
CHAPTER 10	Transactions and concurrency	367
CHAPTER 11	SQL Graph	409
CHAPTER 12	Programmable objects	491
	<i>Appendix: Getting started</i>	<i>527</i>
	<i>Index</i>	<i>547</i>

Contents

<i>Acknowledgments</i>	<i>xxi</i>
<i>About the Author</i>	<i>xxiii</i>
<i>Introduction</i>	<i>xxiv</i>

Chapter 1 Background to T-SQL querying and programming 1

Theoretical background	1
SQL	2
Set theory	3
Predicate logic	4
The relational model	5
Types of database workloads	11
SQL Server architecture	13
On-premises and cloud RDBMS flavors	13
SQL Server instances	15
Databases	16
Schemas and objects	19
Creating tables and defining data integrity	20
Creating tables	21
Defining data integrity	22
Conclusion	26

Chapter 2 Single-table queries 27

Elements of the <i>SELECT</i> statement	27
The <i>FROM</i> clause	29
The <i>WHERE</i> clause	31
The <i>GROUP BY</i> clause	32
The <i>HAVING</i> clause	36
The <i>SELECT</i> clause	37
The <i>ORDER BY</i> clause	42
The <i>TOP</i> and <i>OFFSET-FETCH</i> filters	44
A quick look at window functions	49

Predicates and operators	50
<i>CASE</i> expressions	53
<i>NULLS</i>	56
The <i>GREATEST</i> and <i>LEAST</i> functions	62
All-at-once operations	63
Working with character data	64
Data types	64
Collation	66
Operators and functions	68
The <i>LIKE</i> predicate	81
Working with date and time data	83
Date and time data types	84
Literals	84
Working with date and time separately	88
Filtering date ranges	90
Date and time functions	90
Querying metadata	103
Catalog views	104
Information schema views	105
System stored procedures and functions	105
Conclusion	106
Exercises	107
Exercise 1	107
Exercise 2	107
Exercise 3	108
Exercise 4	108
Exercise 5	109
Exercise 6	109
Exercise 7	109
Exercise 8	110
Exercise 9	110
Exercise 10	111

Solutions	111
Exercise 1	111
Exercise 2	112
Exercise 3	112
Exercise 4	112
Exercise 5	113
Exercise 6	114
Exercise 7	115
Exercise 8	115
Exercise 9	116
Exercise 10	116

Chapter 3 Joins 117

Cross joins	117
SQL-92 syntax	118
SQL-89 syntax	118
Self cross joins	119
Producing tables of numbers	120
Inner joins	121
SQL-92 syntax	121
SQL-89 syntax	122
Inner join safety	123
More join examples	124
Composite joins	124
Non-equi joins	125
Multi-join queries	127
Outer joins	128
Outer joins, described	128
Including missing values	130
Filtering attributes from the nonpreserved side of an outer join	132
Using outer joins in a multi-join query	133
Using the <i>COUNT</i> aggregate with outer joins	136
Conclusion	137

Exercises	137
Exercise 1-1	137
Exercise 1-2	138
Exercise 2	139
Exercise 3	140
Exercise 4	140
Exercise 5	141
Exercise 6	141
Exercise 7	141
Exercise 8	142
Exercise 9	142
Solutions	143
Exercise 1-1	143
Exercise 1-2	143
Exercise 2	144
Exercise 3	144
Exercise 4	145
Exercise 5	145
Exercise 6	145
Exercise 7	146
Exercise 8	146
Exercise 9	147

Chapter 4 Subqueries 149

Self-contained subqueries	149
Self-contained scalar subquery examples	149
Self-contained multivalued subquery examples	151
Correlated subqueries	155
The <i>EXISTS</i> predicate	158
Returning previous or next values	159
Using running aggregates	160
Dealing with misbehaving subqueries	161
<i>NULL</i> trouble	161
Substitution errors in subquery column names	163
Conclusion	166

Exercises	166
Exercise 1	166
Exercise 2	166
Exercise 3	167
Exercise 4	167
Exercise 5	168
Exercise 6	168
Exercise 7	169
Exercise 8	169
Exercise 9	170
Exercise 10	170
Solutions	170
Exercise 1	170
Exercise 2	170
Exercise 3	171
Exercise 4	171
Exercise 5	172
Exercise 6	172
Exercise 7	172
Exercise 8	173
Exercise 9	173
Exercise 10	174

Chapter 5 Table expressions 177

Derived tables	177
Assigning column aliases	179
Using arguments	181
Nesting	181
Multiple references	182
Common table expressions	183
Assigning column aliases in CTEs	184
Using arguments in CTEs	185
Defining multiple CTEs	185
Multiple references in CTEs	186
Recursive CTEs	186

Views	188
Views and the <i>ORDER BY</i> clause	190
View options	192
Inline table-valued functions	196
The <i>APPLY</i> operator	197
Conclusion	200
Exercises	201
Exercise 1	201
Exercise 2-1	201
Exercise 2-2	202
Exercise 3-1	202
Exercise 3-2	203
Exercise 4	203
Exercise 5-1	203
Exercise 5-2	204
Exercise 6-1	205
Exercise 6-2	205
Solutions	206
Exercise 1	206
Exercise 2-1	206
Exercise 2-2	206
Exercise 3-1	207
Exercise 3-2	207
Exercise 4	207
Exercise 5-1	208
Exercise 5-2	208
Exercise 6-1	209
Exercise 6-2	209

Chapter 6 Set operators 211

The <i>UNION</i> operator	212
The <i>UNION ALL</i> operator	213
The <i>UNION (DISTINCT)</i> operator	213

The <i>INTERSECT</i> operator.....	214
The <i>INTERSECT (DISTINCT)</i> operator	215
The <i>INTERSECT ALL</i> operator.....	215
The <i>EXCEPT</i> operator	217
The <i>EXCEPT (DISTINCT)</i> operator.....	218
The <i>EXCEPT ALL</i> operator	219
Precedence	220
Circumventing unsupported logical phases	221
Conclusion	223
Exercises	223
Exercise 1.....	223
Exercise 2	223
Exercise 3	224
Exercise 4	225
Exercise 5	225
Exercise 6	226
Solutions.....	227
Exercise 1.....	227
Exercise 2	227
Exercise 3	228
Exercise 4	228
Exercise 5	229
Exercise 6	230

Chapter 7 T-SQL for data analysis 231

Window functions.....	231
Ranking window functions.....	234
Offset window functions.....	237
Aggregate window functions.....	242
The <i>WINDOW</i> clause	244
Pivoting data.....	246
Pivoting with a grouped query	248
Pivoting with the <i>PIVOT</i> operator.....	249

Unpivoting data	251
Unpivoting with the <i>APPLY</i> operator	253
Unpivoting with the <i>UNPIVOT</i> operator	255
Grouping sets	256
The <i>GROUPING SETS</i> subclause	258
The <i>CUBE</i> subclause	258
The <i>ROLLUP</i> subclause	258
The <i>GROUPING</i> and <i>GROUPING_ID</i> functions	260
Time series	262
Sample data	263
The <i>DATE_BUCKET</i> function	266
Custom computation of start of containing bucket	268
Applying bucket logic to sample data	270
Gap filling	275
Conclusion	280
Exercises	280
Exercise 1	280
Exercise 2	281
Exercise 3	282
Exercise 4	282
Exercise 5	282
Exercise 6	283
Exercise 7	284
Exercise 8	285
Solutions	285
Exercise 1	285
Exercise 2	286
Exercise 3	286
Exercise 4	286
Exercise 5	287
Exercise 6	288
Exercise 7	289
Exercise 8	290

Chapter 8 Data modification 293

Inserting data	293
The <i>INSERT VALUES</i> statement	293
The <i>INSERT SELECT</i> statement	295
The <i>INSERT EXEC</i> statement	296
The <i>SELECT INTO</i> statement	297
The <i>BULK INSERT</i> statement	298
The identity property and the sequence object	298
Deleting data	307
The <i>DELETE</i> statement	308
The <i>TRUNCATE</i> statement	309
<i>DELETE</i> based on a join	310
Updating data	311
The <i>UPDATE</i> statement	313
<i>UPDATE</i> based on a join	314
Assignment <i>UPDATE</i>	316
Merging data	317
Modifying data through table expressions	321
Modifications with <i>TOP</i> and <i>OFFSET-FETCH</i>	324
The <i>OUTPUT</i> clause	326
<i>INSERT</i> with <i>OUTPUT</i>	326
<i>DELETE</i> with <i>OUTPUT</i>	328
<i>UPDATE</i> with <i>OUTPUT</i>	329
<i>MERGE</i> with <i>OUTPUT</i>	330
Nested DML	331
Conclusion	333
Exercises	333
Exercise 1	333
Exercise 2	334
Exercise 3	334
Exercise 4	334
Exercise 5	336
Exercise 6	336

Solutions.....	337
Exercise 1.....	337
Exercise 2.....	338
Exercise 3.....	339
Exercise 4.....	340
Exercise 5.....	340
Exercise 6.....	341

Chapter 9 Temporal tables 343

Creating tables.....	344
Modifying data.....	348
Querying data.....	353
Conclusion.....	360
Exercises.....	360
Exercise 1.....	360
Exercise 2.....	360
Exercise 3.....	361
Exercise 4.....	362
Solutions.....	362
Exercise 1.....	362
Exercise 2.....	364
Exercise 3.....	365
Exercise 4.....	366

Chapter 10 Transactions and concurrency 367

Transactions.....	367
Locks and blocking.....	370
Locks.....	371
Troubleshooting blocking.....	373
Isolation levels.....	380
The <i>READ UNCOMMITTED</i> isolation level.....	381
The <i>READ COMMITTED</i> isolation level.....	382
The <i>REPEATABLE READ</i> isolation level.....	384
The <i>SERIALIZABLE</i> isolation level.....	386

Isolation levels based on row versioning	387
Summary of isolation levels	394
Deadlocks	394
Conclusion	397
Exercises	397
Exercise 1	397
Exercise 2	400
Exercise 3	407

Chapter 11 SQL Graph 409

Creating tables	410
Traditional modeling	411
Graph modeling	417
Querying data	438
Using the <i>MATCH</i> clause	438
Recursive queries	450
Using the <i>SHORTEST_PATH</i> option	454
SQL Graph querying features that are still missing	471
Data modification considerations	474
Deleting and updating data	474
Merging data	477
Conclusion	480
Exercises	481
Exercise 1	481
Exercise 2	482
Exercise 3	483
Exercise 4	483
Solutions	484
Exercise 1	484
Exercise 2	485
Exercise 3	487
Exercise 4	488
Cleanup	490

Chapter 12 Programmable objects	491
Variables	491
Batches	494
A batch as a unit of parsing	494
Batches and variables	495
Statements that cannot be combined in the same batch	495
A batch as a unit of resolution	496
The <i>GO n</i> option	496
Flow elements	497
The <i>IF ... ELSE</i> flow element	497
The <i>WHILE</i> flow element	498
Cursors	500
Temporary tables	505
Local temporary tables	505
Global temporary tables	507
Table variables	508
Table types	509
Dynamic SQL	510
The <i>EXEC</i> command	511
The <i>sp_executesql</i> stored procedure	511
Using <i>PIVOT</i> with Dynamic SQL	512
Routines	513
User-defined functions	514
Stored procedures	515
Triggers	517
Error handling	521
Conclusion	525
<i>Appendix: Getting started</i>	527
<i>Index</i>	547

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 *TSQVL6* 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*.

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.

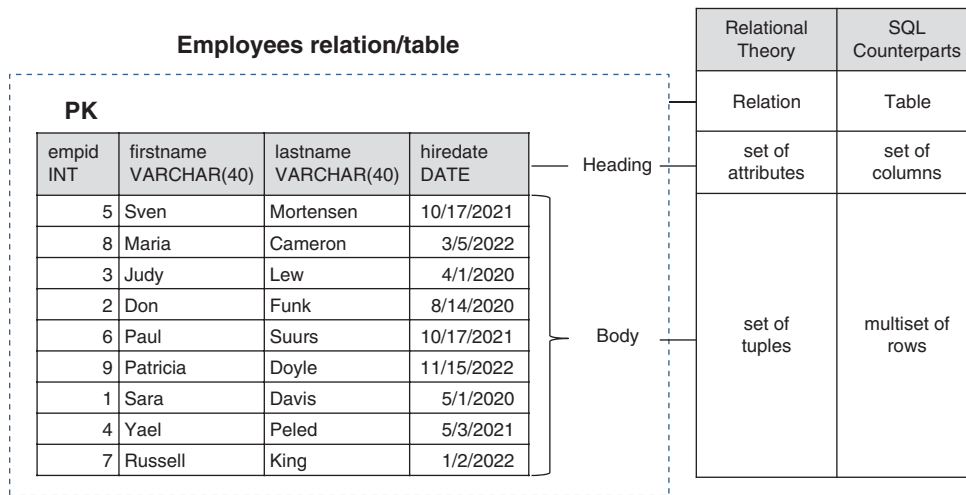


FIGURE 1-1 Illustration of *Employees* relation

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 *NULL*s). 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*.

Orders	
PK	<u>orderid</u>
PK	<u>productid</u>
	orderdate qty customerid companyname

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*.

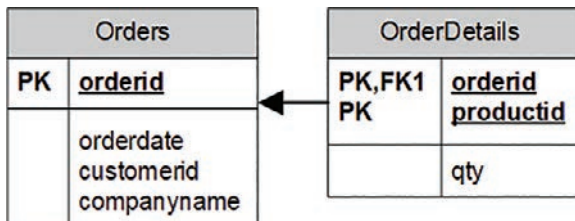


FIGURE 1-3 Data model after applying 2NF and before 3NF

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.

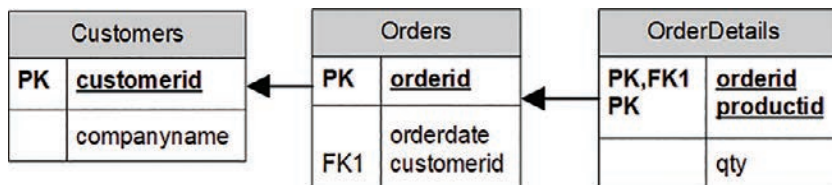


FIGURE 1-4 Data model after applying 3NF

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.

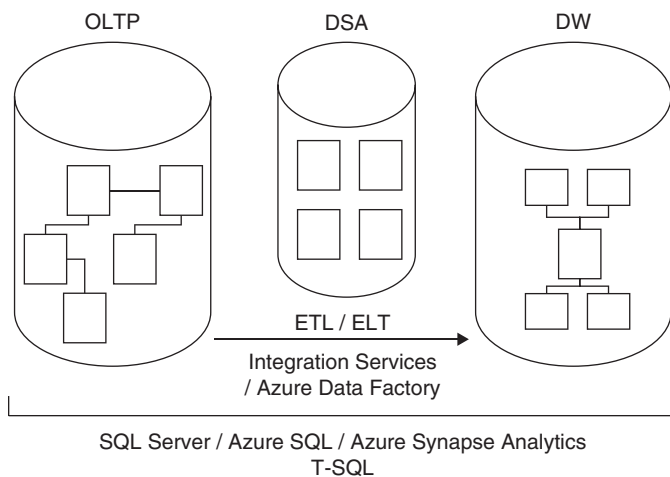


FIGURE 1-5 Classes of database systems

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.

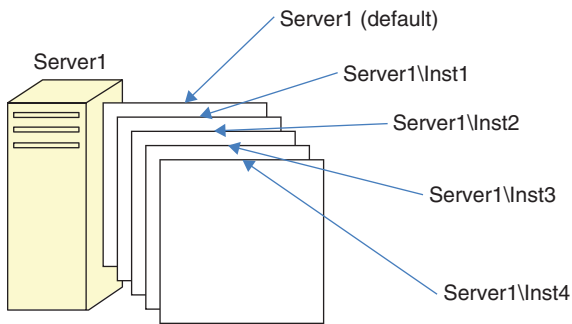


FIGURE 1-6 Multiple instances of SQL Server on the same computer

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.

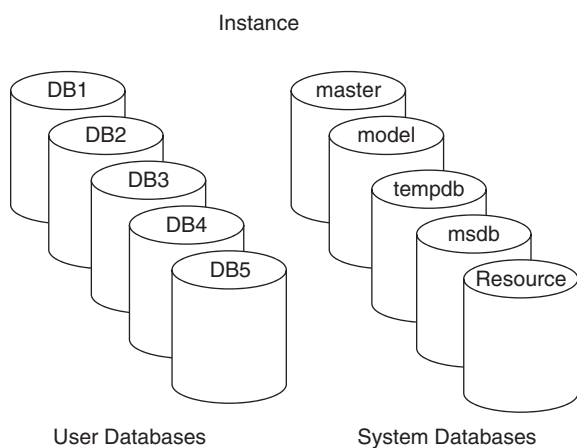


FIGURE 1-7 An example of multiple databases on a SQL Server instance

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.

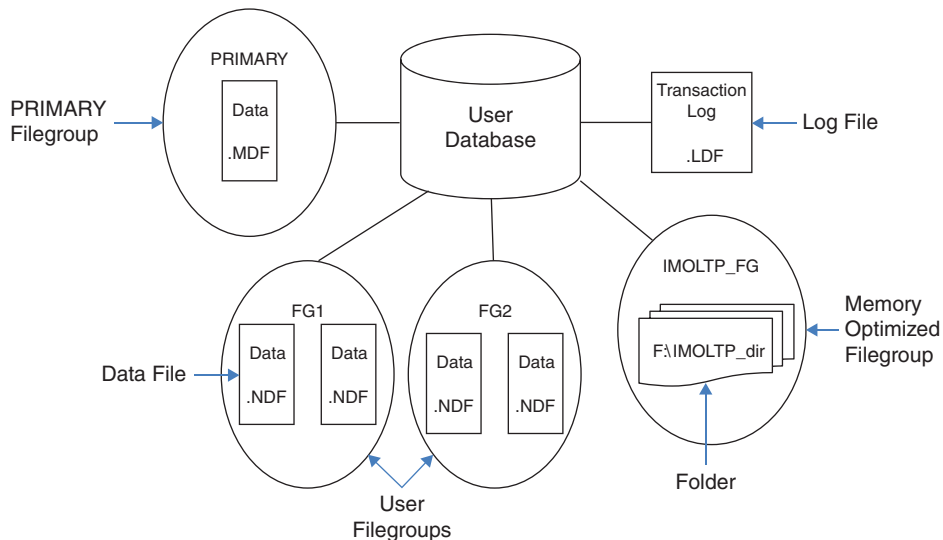


FIGURE 1-8 Database layout

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.

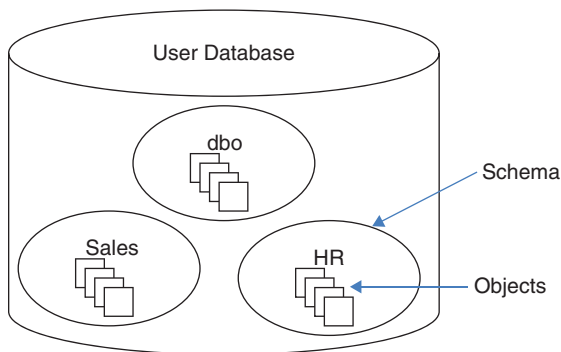


FIGURE 1-9 A database, schemas, and database objects

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 *NULL*s 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 *NULL*s, 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 *NULL*s from the index, so duplicate *NULL*s 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

- + (addition) operator, 50–53
- = (assignment) operator, 50–53
- * (asterisk), SELECT and, 41–42, 159
- \ (backslash), 15
- ^ (caret), 83
- [<character>-<character>] wildcard, 83
- , (comma)
 - in cross joins, 118
 - in grouping sets, 258
- + (concatenation) operator, 69–70
- / (division) operator, 50–53
- " (double quotes), 68
- \$edge_id
 - metadata, querying, 434–438
 - overview of, 419
- \$from_id
 - DELETE and, 474–477
 - edge table creation and, 422–434
 - MATCH and, 441
 - metadata, querying, 434–438
 - overview of, 419
- \$from_node, 419
- > (greater than) operator, 50–53
- >= (greater than or equal to) operator, 50–53
- @@identity, 300
- <> (inequality) operator, 50–53
- < (less than) operator, 50–53
- [<list of characters>] wildcard, 82
- % (modulo) operator, 50–53
- * (multiplication) operator, 50–53
- @nextval, 317
- \$node_id, 435
 - overview of, 419
 - querying, 421
- != (not equal to) operator, 50–53
- !> (not greater than) operator, 50–53
- !< (not less than) operator, 50–53

- () (parentheses)
 - in grouping sets, 258
 - set operators and, 221
- % (percent) wildcard, 81–82
- # (pound sign), 507
- ;(semicolon), 22, 29, 319
- ' (single quotes), 68
- @@SPID, 375
- [] (square brackets), 68, 82
- (subtraction) operator, 50–53
- \$to_id
 - DELETE and, 474–477
 - edge table creation and, 422–434
 - MATCH and, 441
 - metadata, querying, 434–438
 - overview of, 419
- \$to_node, 419
- @@TRANCOUNT, 368
- _ (underscore), 82
- 1NF (first normal form), 9
- 2NF (second normal form), 9–10
- 3NF (third normal form), 10–11

A

- Accelerated Database Recovery (ADR), 369, 387
- ACID (atomicity, consistency, isolation, and durability), 368–370
- ADD CONSTRAINT UNQ, 424
- addition (+) operator, 50–53
- ADO.NET, 494
- AFTER INSERT, 519
- after triggers, 518–519
- aggregate functions. *See also* COUNT; pivoting data; window functions
 - graph path aggregate functions, 456

window functions

- window functions (*continued*)
 - NULLs and, 35
 - overview of, 231–233, 242–244
 - running aggregates, subqueries and, 160–161
 - aliases
 - in common table expressions, 184
 - cross joins and, 118
 - derived tables and, 179–181
 - inline table-valued functions (TVFs), 196–197
 - INSERT VALUES and, 295
 - ORDER BY and, 42–44
 - SELECT clause and, 37–42
 - self joins and, 119
 - ALL
 - EXCEPT ALL, 219–220
 - INTERSECT ALL, 215–217
 - temporal table queries, 358
 - UNION ALL, 213
 - all-at-once operations, 63–64
 - ALTER DATABASE, 68
 - ALTER PROC, 515
 - ALTER SEQUENCE, 303
 - ALTER TABLE, 24, 194
 - data integrity, defining, 22
 - lockable resources types and, 373
 - MATCH and, 442–444
 - sequence objects and, 306
 - temporal table creation, 344–348, 353
 - ALTER TABLE ADD CONSTRAINT, 309
 - ALTER TABLE DROP CONSTRAINT, 309
 - ALTER VIEW, 189, 193
 - alternate keys, 8
 - American National Standards Institute (ANSI), 2–3
 - AND, 51
 - MATCH and, 446–448
 - MERGE and, 321
 - temporal table queries, 357
 - anonymous result columns, 37
 - ANSI (American National Standards Institute), 2–3
 - application programming interfaces (APIs), 494
 - application-time period tables, 343
 - APPLY, 63
 - exercise descriptions, 201–206
 - exercise solutions, 206–209
 - table expressions, 197–200
 - unpivoting data with, 253–255
 - arbitrary length pattern, 455, 471–473
 - arguments
 - in common table expressions, 185
 - derived table queries and, 181
 - arithmetic operators, 50–53
 - AS
 - data types and, 302
 - derived tables and, 177–178, 180, 182
 - SELECT and, 37
 - AS EDGE, 422–434
 - AS NODE, 420–422
 - AS OF, 355, 358
 - ASC, ORDER BY and, 42–44
 - assignment (=) operator, 50–53
 - assignment UPDATE, 316–317
 - asterisk (*)
 - multiplication operator, 50–53
 - SELECT and, 41–42, 159
 - AT TIME ZONE, 94–96, 358–359
 - atomicity, 368
 - auditing
 - with data definition language (DDL) triggers, 519–521
 - with data manipulation language (DML) triggers, 518–519
 - authenticated login, 17–18
 - AVG, 35, 456
 - Azure Data Factory, 13
 - Azure Data Studio (ADS), 494
 - Azure SQL Database, 1–2, 14–15
 - collation, 66–68
 - databases, 17–19
 - edge constraints, 423
 - gap filling, 276
 - INSERT VALUES and, 294–295
 - ledgers, 353
 - online transactional processing (OLTP), 11–13
 - SHORTEST_PATH option, 454–464
 - transactions
 - isolation levels and, 380, 388, 392
 - lock modes and compatibility, 371–372
 - locks and blocking, 370–371
 - Azure SQL Managed Instance, 67
 - Azure Synapse Analytics, 1, 15
- ## B
- backslash (\), 15
 - bag, 40
 - batches
 - GO n option, 496
 - overview of, 494
 - statements that cannot be combined in, 495–496

- transactions versus, 494
 - as unit of parsing, 494–495
 - as unit of resolution, 496
 - variables and, 495
 - BEGIN, 498
 - BEGIN TRAN (TRANSACTION), 351, 367
 - BEGIN TRY, 521–525
 - BETWEEN
 - temporal table queries, 357, 358
 - use of, 50–53
 - BIGINT data type, 302, 421
 - binding, 496
 - bitemporal tables, 343
 - blocking, 373–380. *See also* isolation levels; locks; transactions
 - blocking_session_id, 379
 - BREAK, 499
 - bucketized data
 - DATE_BUCKET
 - bucket logic, applying to data, 270–275
 - containing bucket, computation of start of, 268–270
 - overview of, 266–268
 - exercise descriptions, 280–285
 - exercise solutions, 285–291
 - gap filling, 275–279
 - bugs, subqueries and
 - NULL trouble, 160–161
 - substitution errors in subquery column names, 163–164
 - BULK INSERT, 298
- C**
- CACHE, sequence objects and, 303
 - caching, 303
 - candidate keys, 8
 - Cantor, Georg, 3
 - caret (^), 83
 - CASCADE, 25, 423
 - CASE expressions
 - all-at-once operations and, 63–64
 - GREATEST/LEAST and, 62–63
 - overview of, 53–56, 147
 - pivoting with grouped queries, 248–249
 - temporal table queries, 359
 - CAST, 211
 - correlated subqueries and, 157
 - date and time functions, 91–93
 - date and time literals, 87–88
 - REPLICATE and, 74–75
 - catalog views, 104–105
 - CATCH, error handling, 521–525
 - change management, DDL triggers and, 519–521
 - CHAR, 51, 64–66
 - character data, working with. *See also* functions; operators
 - CHARINDEX, 72
 - collation, 66–68
 - COMPRESS/DECOMPRESS, 79
 - CONCAT, 69–70
 - CONCAT_WS, 69–70
 - concatenation, 69–70
 - data types, overview of, 64–66
 - DATALENGTH, 71–72
 - FORMAT, 78
 - LEFT, 71
 - LEN, 71–72
 - LIKE, 81–83
 - LOWER, 75
 - PATINDEX, 72
 - REPLACE, 72–73
 - REPLICATE, 74–75
 - RIGHT, 71
 - RTRIM and LTRIM, 76
 - STRING_AGG, 81
 - STRING_SPLIT, 80
 - STUFF, 75
 - SUBSTRING, 70–71
 - TRANSLATE, 73–74, 77
 - TRIM, 76–78
 - UPPER, 75
 - [<character>-<character>] wildcard, 83
 - CHARINDEX, 72
 - CHECK, 25–26, 56
 - CASE expressions, 53–56
 - identity property and, 299, 300
 - working with date and time separately, 88–90
 - CHECK OPTION, views and, 194–195
 - CHOOSE, CASE expressions and, 55–56
 - CLOSE, WHILE flow element and, 504
 - closed-world assumption (CWA), 6
 - closure property, 5, 39
 - cloud computing, 14–15
 - COALESCE, 55–56, 70
 - Codd, Edgar F.4, 5, 7
 - coding style, 22
 - COLLATE, 66–68
 - collation, 17, 66–68

column names

- column names
 - aliases, assigning
 - in common table expressions, 184
 - in derived tables, 179–181
 - derived tables and, 179
- COLUMNPROPERTY, 106
- comma (,)
 - in cross joins, 118
 - in grouping sets, 258
- COMMIT TRAN (TRANSACTION), 351
 - isolation levels and, 383, 389, 393
 - lock modes and compatibility, 371
 - overview of, 367
 - troubleshooting blocking and, 374
- common table expressions (CTEs)
 - arguments, use of, 185
 - column aliases, assigning, 184
 - defining multiple, 185
 - exercise descriptions, 201–206
 - exercise solutions, 206–209
 - multiple references in, 186
 - overview of, 183–184
 - recursive, 186–188
- commutative relationships, 411
- comparison operators, 50–53
- compatible data types, 211
- composite constraints, 22–23
- composite joins
 - exercise descriptions, 137–143
 - exercise solutions, 143–147
 - use and syntax, 124–125
- COMPRESS, 79
- compression, 66
- CONCAT, 69–70
- CONCAT_WS, 69–70
- concatenation, 69–70
- concurrency, transactions and, 371–372.
- See also* transactions
- conflict detection, SNAPSHOT and, 390–392
- conjunction of predicates, 50
- consistency, 368
- constraints, 22–26
 - check, 25–26, 89
 - composite, 22–23
 - default, 26
 - edge, 423
 - foreign key, 8, 24–25
 - overview of, 8
 - primary key, 9, 23
 - unique, 23–24, 61
- constructors, row, 316
- contained databases, 18
- containing bucket, computation of start of, 268–270
- CONTINUE, 500
- CONVERT, 87–88, 89, 91–93, 211
- correlated subqueries
 - exercise descriptions, 166–170
 - exercise solutions, 170–175
 - overview of, 155–157
- COUNT, 180, 456, 463
 - HAVING clause and, 36
 - with outer joins, 144
 - outer joins and, 136–137
 - SELECT clause and, 39
 - subqueries and, 170–171
 - WHERE clause and, 34–35
- CREATE DATABASE, DDL triggers and, 519–521
- CREATE DEFAULT, 495
- CREATE FUNCTION, 495
- CREATE OR ALTER VIEW, 193
- CREATE PROCEDURE, 495
- CREATE RULE, 495
- CREATE SCHEMA, 495
- CREATE SEQUENCE, 302–307
- CREATE TABLE, 21
 - data integrity, defining, 22
 - DDL triggers, 519–521
 - edge tables, creating, 422–434
 - INSERT VALUES and, 294
 - node tables, creating, 420–422
- CREATE TRIGGER, 495
- CREATE UNIQUE INDEX, 24
- CREATE VIEW, 189, 495
- CROSS APPLY
 - exercise descriptions, 205–206
 - exercise solutions, 209
 - overview of, 197–200
 - unpivoting data, 253–255
- cross joins
 - exercise descriptions, 137–143
 - exercise solutions, 143–147
 - use and syntax, 117–121
- CUBE, 258
- CURRENT, 371
- current date and time functions, 90–91
- CURRENT ROW, 233
- CURRENT_TIMESTAMP, 90–91
- cursors, 43, 500–505
- CYCLE, sequence objects and, 303

D

- Darwen, Hugh, 5
- data analysis. *See* grouping sets; pivoting data;
- time series data; window functions
- data compression, 66
- data control language (DCL), 3
- data definition language (DDL), 3
 - batches, 496
 - triggers, 519–521
- data integrity, defining, 22–26. *See also* constraints
- data manipulation language (DML), 3
 - batches, 496
 - nested DML, 331–333
 - triggers, 518–519
- data marts, 12
- data types
 - BIGINT, 302, 421
 - CHAR, 51, 64–66
 - compatible, 211
 - definition of, 7
 - INT, 21, 52, 491
 - NCHAR, 51, 64–66
 - NUMERIC, 52, 157
 - NVARCHAR, 51, 64–66, 81
 - overview of, 64–66
 - precedence of, 52
 - SQL_VARIANT, 307, 507
 - VARCHAR, 21, 51, 64–66, 74, 81, 85, 294
- data warehouses (DWs), 12–13
- database administrators (DBAs), 17–18
- database triggers, 519–521
- database users, 18
- DATABASEPROPERTYEX, 106
- databases, SQL Server, 16–19. *See also* RDBMS (relational database management systems)
 - architecture and layout, 16–19
 - contained, 18
 - database system types, 11–13
 - file extensions, 19
 - schemas and objects, 19–20
 - tempdb
 - local temporary tables, 505–507
 - table variables, 508–509
- DATALength, 71–72
- data-staging area (DSA), 13
- data-type precedence, 85
- DATE, 21
 - bucketized data and, 274
 - data types, 84
 - literals, 84–88
 - working with date and time separately, 88–90
- Date, Chris, 5
- date and time data, working with. *See also* time series data
 - AT TIME ZONE, 94–96
 - CAST, 91–93
 - CONVERT, 91–93
 - current date and time functions, 90–91
 - CURRENT_TIMESTAMP, 90–91
 - data ranges, filtering, 90
 - data types, 84
 - DATEADD, 96, 131, 274–275
 - DATEDIFF, 96–98, 131, 143, 268–270, 274–275
 - DATEDIFF_BIG, 96–98
 - DATENAME, 99
 - DATEPART, 98
 - DATETRUNC, 99–100
 - EOMONTH, 101–102
 - FROMPARTS, 101
 - GENERATE_SERIES, 102–103
 - GETDATE, 90–91
 - GETUTCDATE, 90–91
 - ISDATE, 100
 - literals, 84–88
 - PARSE, 91–93
 - SWITCHOFFSET, 93–94
 - SYSDATETIME, 90–91
 - SYSDATETIMEOFFSET, 90–91
 - SYSUTCDATETIME, 90–91
 - TODATETIMEOFFSET, 94
 - TRY_CAST, 91–93
 - TRY_CONVERT, 91–93
 - TRY_PARSE, 91–93
 - working with separately, 88–90
 - YEAR, MONTH, and DAY, 98–99
- DATE_BUCKET, 100
 - bucket logic, applying to data, 270–275
 - containing bucket, computation of start of, 268–270
 - exercise descriptions, 280–285
 - exercise solutions, 285–291
 - gap filling, 275–279
 - overview of, 266–268
- DATEADD, 96, 131, 274–275
- DATEDIFF, 96–98, 131, 143, 175, 268–270, 274–275
- DATEDIFF_BIG, 96–98
- DATEFORMAT, 84–88
- DATEFROMPARTS, 101
- DATENAME, 99
- DATEPART, 98

DATETIME

- DATETIME
 - data types, 84
 - literals, 88–90
 - temporal table creation, 344–348
- DATETIME2
 - data types, 84
 - literals, 84–88
 - temporal table creation, 344–348
- DATETIME2FROMPARTS, 101
- DATETIMEFROMPARTS, 101
- DATETIMEOFFSET, 93–94
 - data types, 84
 - literals, 84–88
- DATETIMEOFFSETFROMPARTS, 101
- DATETRUNC, 99–100
- DAY, 98–99
- DBCC CHECKIDENT, 302
- DCL (data control language), 3
- DDL. *See* data definition language (DDL)
- DEADLOCK_PRIORITY, 394
- deadlocks, 394–397
- deadly embrace deadlocks, 396
- DEALLOCATE, WHILE flow element and, 504
- declarative data integrity, 22
- DECLARE
 - table types, 509
 - table variables, 508–509
 - variables and, 491–493
- DECLARE CURSOR, 503
- DECOMPRESS, 79
- default constraints, 26
- default instances, 15
- degenerate intervals, 351–352
- DELETE
 - DML triggers, 518–519
 - exercise descriptions, 333–337
 - exercise solutions, 337–341
 - joins and, 310–311
 - MERGE and, 317–321
 - OUTPUT and, 328–329
 - overview of, 308
 - row-versioning isolation and, 388
 - SQL Graph and, 474–477
 - table expressions, modifying data through, 321–324
 - temporal table modification, 348–353
 - TOP filter in, 324–326
- deleting data. *See also* UPDATE
 - DELETE
 - joins and, 310–311
 - overview of, 308
 - sample data for, 307–308
 - SQL Graph, 474–477
 - TRUNCATE, 309–310
- DENSE_RANK, 234–237
- derived tables
 - arguments, use of, 181
 - column aliases, assigning, 179–181
 - exercise descriptions, 201–206
 - exercise solutions, 206–209
 - multiple references, 182–183
 - nesting, 181–182
 - overview of, 177–179
- DESC, ORDER BY and, 42–44
- DESCENDING, 459
- dictionary sorting, 66
- dirty reads, 381–382
- disjunction of predicates, 50
- DISTINCT, 211–212
 - aggregate functions and, 35
 - EXCEPT (DISTINCT), 218–219
 - INTERSECT (DISTINCT), 215
 - IS DISTINCT FROM, 57–61
 - IS NOT DISTINCT FROM, 57–61
 - ORDER BY and, 42–44
 - outer joins and, 147
 - ranking window functions and, 236
 - SELECT and, 41
 - self-contained multivalued subqueries and, 154, 171
 - UNION (DISTINCT), 213–214
- distinct, definition of, 4
- division (/) operator, 50–53
- DML. *See* data manipulation language (DML)
- double quotes ("), 68
- DROP, 309
- DROP CONSTRAINT, 306
- DROP FUNCTION IF EXISTS, 197
- DROP TABLE IF EXISTS, 21, 24, 26
 - global temporary tables, 508
 - local temporary tables, 507
 - self-contained multivalued subqueries and, 155
 - sequence objects and, 304, 307
 - subqueries, NULL trouble and, 164
 - UNPIVOT and, 256
 - UPDATE and, 316–317
- DROP VIEW, 495–496
- durability, transactions, 369
- DWs (data warehouses), 12–13
- dynamic management views (DMV), 374
 - sys.dm_exec_connections, 376
 - sys.dm_exec_input_buffer, 377

- sys.dm_exec_requests, 377–379
- sys.dm_exec_sessions, 378
- sys.dm_exec_sql_text, 377
- sys.dm_os_waiting_tasks, 379
- sys.dm_tran_locks, 374–376
- dynamic SQL
 - definition of, 510
 - EXEC command, 511
 - overview of, 510
 - PIVOT operator, 512–513

E

- edge constraints, 423
- edge tables, creating, 422–434
- EDGE_ID_FROM_PARTS, 437–438
- edges, 417
- ELSE
 - CASE expressions and, 53–56
 - IF . . . ELSE flow element, 497–498
- ELSE NULL, 248
- ENCRYPTION, views and, 192–193
- END, 498
- END TRY, 521–525
- entity relationship modeling (ERM), 8
- EOMONTH, 98, 101–102
- equi joins
 - exercise descriptions, 137–143
 - exercise solutions, 143–147
 - overview of, 125
- error handling
 - deadlocks and, 394–397
 - programmable objects and, 521–525
 - transactions and, 370
- ERROR_LINE, 522
- ERROR_MESSAGE, 522
- ERROR_NUMBER, 522–523
- ERROR_PROCEDURE, 522
- ERROR_SEVERITY, 522
- ERROR_STATE, 522
- ESCAPE, 83
- EVENTDATA, 520
- EXCEPT
 - exercise descriptions, 223–226
 - exercise solutions, 227–230
 - MATCH and, 449–450
 - precedence and, 220–221
 - SELECT INTO and, 297
 - use of, 217–220
- EXCEPT (DISTINCT), 218–219

- EXCEPT ALL, 219–220
- exclusive locks, 371–372
- EXEC, 296–297, 511
- exercise descriptions. *See also* exercise solutions
 - DELETE, 333–337
 - INSERT, 333–337
 - JOIN, 137–143
 - MERGE, 333–337
 - OFFSET-FETCH, 333–337
 - OUTPUT, 333–337
 - queries, 107–111
 - SELECT, 107–111
 - set operators, 223–226
 - SQL Graph, 481–483
 - subqueries, 166–170
 - table expressions, 201–206, 333–337
 - temporal tables, 360–362
 - TOP filter, 333–337
 - transactions, 397–408
 - window functions, 280–285
- exercise solutions
 - DELETE, 337–341
 - INSERT, 337–341
 - JOIN, 143–147
 - MERGE, 337–341
 - OFFSET-FETCH, 337–341
 - OUTPUT, 337–341
 - queries, 111–116
 - SELECT, 111–116
 - set operators, 227–230
 - SQL Graph, 484–489
 - subqueries, 170–175
 - table expressions, 206–209, 337–341
 - temporal tables, 362–366
 - TOP filter, 337–341
 - window functions, 285–291
- EXISTS
 - correlated subqueries and, 158–159, 172–173
 - MATCH and, 449–450
 - subqueries, NULL trouble and, 161–163, 170–175
- external aliasing, 180–181
- extract, load, and transform (ELT), 13
- extract, transform, and load (ETL), 13

F

- FALSE, 50
 - all-at-once operations, 63–64
 - HAVING clause and, 36
 - IF . . . ELSE flow element and, 497–498

FALSE

FALSE (*continued*)

- NULLs, overview of, 56–61
 - subqueries, NULL trouble and, 161–163, 173
 - WHERE clause and, 32
 - WHILE flow element and, 498
- FETCH, 191
- OFFSET-FETCH, 47–48
 - WHILE flow element and, 503
- filegroups, 19
- filters. *See also* HAVING; WHERE
- date ranges, 90
 - OFFSET-FETCH
 - data modification with, 324–326
 - derived tables and, 178
 - exercise descriptions, 333–337
 - exercise solutions, 337–341
 - overview of, 47–48
 - views and ORDER BY clause, 222
 - predicates, 129–130
 - TOP
 - data modification with, 324–326
 - derived tables and, 178
 - exercise descriptions, 333–337
 - exercise solutions, 337–341
 - overview of, 44–47
 - subqueries and, 170–171
 - views and ORDER BY clause, 191–192, 222
 - WITH TIES, 46–47, 48, 170–171
- FIRST_VALUE, 237–242
- flow elements
- IF . . . ELSE, 497–498
 - WHILE, 498–500
- fn_helpcollations, 66–68
- follow relationships, 411–412
- FOR PATH, 455
- FOR SYSTEM_TIME CONTAINED IN, 357
- FOR SYSTEM_TIME, 351–352, 353–359
- foreign key constraints, 8, 24–25
- FORMAT, 75, 78
- friendship relationships, 411, 468
- FROM. *See also* JOIN
- DELETE and, 308, 310
 - derived tables and, 182
 - logical query processing order, 117
 - MATCH and, 444–448
 - overview of, 29–30
 - PIVOT and, 249–251
 - SHORTEST_PATH and, 455–456
 - with table expressions
 - common table expressions (CTEs), 186
 - derived tables, 177, 180
 - temporal table queries, 356, 358
 - UNPIVOT and, 255–256
 - unpivoting data and, 255
 - user-defined functions (UDFs) and, 514
 - WINDOW clause, 244–246
- from nodes, 411, 417
- FROMPARTS functions, 101
- FULL, 128
- functions. *See also* aggregate functions; window functions; *specific function names*
- for character string operations
 - CHARINDEX, 72
 - COMPRESS/DECOMPRESS, 79
 - CONCAT/CONCAT_WS, 69–70
 - DATALLENGTH, 71–72
 - FORMAT, 78
 - LEFT, 71
 - LEN, 71–72
 - LOWER, 75
 - PATINDEX, 72
 - REPLACE, 72–73
 - REPLICATE, 74–75
 - RIGHT, 71
 - RTRIM and LTRIM, 76
 - STRING_AGG, 81
 - STRING_SPLIT, 80
 - STUFF, 75
 - SUBSTRING, 70–71
 - TRANSLATE, 73–74, 77
 - TRIM, 76–78
 - UPPER, 75
 - date and time
 - AT TIME ZONE, 94–96
 - CAST, 91–93
 - CONVERT, 91–93
 - current date and time functions, 90–91
 - CURRENT_TIMESTAMP, 90–91
 - DATEADD, 96
 - DATEDIFF, 96–98
 - DATEDIFF_BIG, 96–98
 - DATENAME, 99
 - DATEPART, 98
 - DATETRUNC, 99–100
 - EOMONTH, 101–102
 - FROMPARTS, 101
 - GENERATE_SERIES, 102–103
 - GETDATE, 90–91
 - GETUTCDATE, 90–91
 - ISDATE, 100
 - PARSE, 91–93
 - SWITCHOFFSET, 93–94

- SYSDATETIME, 90–91
- SYSDATETIMEOFFSET, 90–91
- SYSUTCDATETIME, 90–91
- TODATETIMEOFFSET, 94
- TRY_CAST, 91–93
- TRY_CONVERT, 91–93
- TRY_PARSE, 91–93
- YEAR, MONTH, and DAY, 98–99
- overview of, 68–69
- system stored, 105–106
- user-defined, 514–515

G

- gap filling, 275–279
- GENERATE_SERIES, 102–103, 276–279
- GENERATED ALWAYS AS ROW END, 344–348, 363
- GENERATED ALWAYS AS ROW START, 344–348, 363
- GETDATE, 90–91
- GETUTCDATE, 90–91
- global temporary tables, 507–508
- globally unique identifiers (GUIDs), 296, 514
- GO, 189, 496–497
 - batches and, 496–497
 - variables and, 494
- graph data, querying
 - exercise descriptions, 481–483
 - exercise solutions, 484–489
 - features that are still missing, 471–473
 - LAST_NODE function, 464–471
 - MATCH clause, 438–450, 477–480
 - overview of, 438
 - recursive queries, 450–454
 - SHORTEST_PATH option, 454–464
- graph modeling. *See also* SQL Graph
 - edge tables, creating, 422–434
 - exercise descriptions, 481–483
 - exercise solutions, 484–489
 - metadata, querying, 434–438
 - node tables, creating, 420–422
 - overview of, 417–419
- graph path aggregate functions, 456
- graph_id, 435, 436
- GRAPH_ID_FROM_EDGE_ID, 437–438
- GRAPH_ID_FROM_NODE_ID, 437–438
- GREATEST, 36, 62–63
- GROUP BY. *See also* grouping sets
 - CUBE, 258
 - GROUPING SETS, 258
 - NULL and, 61
 - overview of, 31–36
 - pivoting data with, 248–249
 - ROLLUP subclause, 258–259
 - with table expressions, 179–181, 182
 - WINDOW, 244–246
- grouped queries, pivoting data with, 248–249
- GROUPING, 260–262
- GROUPING SETS, 258
- grouping sets. *See also* GROUP BY
 - CUBE, 258
 - exercise descriptions, 280–285
 - exercise solutions, 285–291
 - GROUPING and GROUPING_ID, 260–262
 - GROUPING SETS, 258
 - overview of, 256–257
 - ROLLUP, 258–259
- GROUPING_ID, 260–262
- GZIP algorithm, 79

H

- HAVING
 - CASE expressions, 53–56
 - overview of, 36–37
 - WINDOW clause, 244–246
- hidden columns, 355
- historical data. *See* temporal tables
- history retention policy, 344
- HISTORY_RETENTION_PERIOD, 344
- HOLDLOCK, 381

I

- IaaS (infrastructure as a service), 14
- IDENT_CURRENT, 300–302
- identifier names, delimiting, 31
- identifiers, quoted, 68
- identity property, 298–302
- IDENTITY_INSERT, 301–302
- IF . . . ELSE flow element, 497–498
- IGNORE NULLS, 239–242
- IIF, 55–56
- IMPLICIT_TRANSACTIONS, 367–368
- IN
 - PIVOT operator and, 512
 - self-contained multivalued subqueries and, 151–155
 - subqueries, NULL trouble and, 161–163, 173
 - use of, 50–53

inconsistent analysis

- inconsistent analysis, 384, 393
- INCREMENT BY, sequence objects and, 303
- indentation, recursive queries and, 453–454
- indexes, unique, 23
- indirect friendship relationships, 455
- information schema views, 105
- INFORMATION_SCHEMA, 105
- infrastructure as a service (IaaS), 14
- inline aliasing, 180–181
- inline table-valued functions (TVFs)
 - exercise descriptions, 201–206
 - exercise solutions, 206–209
 - overview of, 196–197
- In-Memory OLTP, 18–19, 367
- inner joins
 - exercise descriptions, 137–143
 - exercise solutions, 143–147
 - MATCH and, 438–450
 - use and syntax, 121–123
- inner queries, 178–179
- INSERT
 - BULK INSERT, 298
 - cross joins and, 120
 - DML triggers, 518–519
 - exercise descriptions, 333–337
 - exercise solutions, 337–341
 - identity property, 298–302
 - MERGE and, 317–321, 478
 - NEXT VALUE FOR and, 303–307
 - OUTPUT and, 326–327
 - self-contained multivalued subqueries and, 154
 - table expressions, modifying data through, 321–324
 - temporal table modification, 348–353
 - TOP filter in, 324–326
 - transactions and, 367
- INSERT DEFAULT VALUES, 497
- INSERT EXEC, 296–297
- INSERT INTO, 507
 - edge tables, creating, 424–426
 - identity property and, 298–302
 - INSERT INTO, 507
 - node tables, creating, 420–422
- INSERT SELECT, 295–296, 332–333
- INSERT VALUES, 293–295
- inserting data, 298–302. *See also* INSERT
 - BULK INSERT, 298
 - identity property and, 298–302
 - INSERT EXEC, 296–297
 - INSERT INTO, 298–302, 420–422, 424–426
 - INSERT SELECT, 295–296
 - INSERT VALUES, 293–295
 - SELECT INTO, 297–298
 - sequence objects, 302–307
- instances, SQL Server, 15–16
- instead of triggers, 518–519
- INT, 21, 52, 491
- integrity rule enforcement
 - with data definition language (DDL) triggers, 519–521
 - with data manipulation language (DML) triggers, 518–519
- International Organization for Standardization (ISO), 2–3
- INTERSECT
 - exercise descriptions, 223–226
 - exercise solutions, 227–230
 - precedence and, 220–221
 - use of, 214–217
- INTERSECT (DISTINCT), 215
- INTERSECT ALL, 215–217
- INTO
 - INSERT INTO, 298–302
 - INSERT VALUES and, 294
 - MERGE and, 478
 - OUTPUT and, 326, 329
 - SELECT INTO, 297–298
- IS DISTINCT FROM, 57–61
- IS NOT DISTINCT FROM, 57–61
- IS NOT NULL
 - overview of, 56–61
 - subqueries, NULL trouble and, 163
- IS NULL, 55–61, 130, 132–133, 498
- ISDATE, 100
- ISO (International Organization for Standardization), 2–3
- isolation levels. *See also* row versioning
 - exercises, 397–408
 - overview of, 368–369, 380–381, 387–388
 - READ COMMITTED, 382–384
 - READ UNCOMMITTED, 381–382
 - REPEATABLE READ, 384–385
 - SERIALIZABLE, 386–387
- SNAPSHOT
 - conflict detection, 390–392
 - isolation, overview of, 388–390
 - READ COMMITTED SNAPSHOT, 392–393
 - summary of, 394

J

JOIN

- composite joins, 124–125
- cross joins, 117–121
- DELETE and, 310–311
- derived tables, multiple references, 182–183
- equi joins, 125
- exercise descriptions, 137–143
- exercise solutions, 143–147
- inner joins, 121–123
- MATCH and, 438–450
- multi-join queries, 127
- natural joins, 125
- non-equi joins, 125–127
- outer joins, 219
 - COUNT aggregate and, 136–137
 - description of, 128–130
 - filtering attributes from nonpreserved side of, 132–133
 - missing values, inclusion of, 130–132
 - in multi-join queries, 133–135
- overview of, 117
- self joins, 119
- theta joins, 125
- UPDATE and, 314–316

JSON (JavaScript Object Notation), 491

K

keys

- alternate, 8
- candidate, 8
- foreign, 8, 25
- generation of
 - identity property, 298–302
 - sequence objects, 302–307
- primary, 9
- surrogate, 298–302

KILL, 379–380

L

- LAG, 160, 237–242, 509
- language independence, 2
- LANGUAGE/DATEFORMAT, 84–88
- large object (LOB), 66
- LAST_NODE, 464–471
- LAST_VALUE, 237–242, 456, 460, 464–471

- LATERAL, 197
- .ldf file extension, 19
- LEAD, 160, 237–242
- LEADING, 78
- LEAST, 36, 62–63
- ledgers, 353
- LEFT, 71, 128
- LEN, 71–72
- LIKE, 50–53, 81–83
- linked history tables, 344–348
- [<list of characters>] wildcard, 82
- literals, data and time data, 84–88
- local temporary tables, 505–507
- LOCK_ESCALATION, 373
- LOCK_TIMEOUT, 379–380
- locks
 - exercises, 397–408
 - lockable resources types, 372–373
 - modes and compatibility, 371–372
 - overview of, 370–371
- logical operators, 50–53
- logical query processing, 117. *See also* set operators
- logins, SQL Server database, 17–18
- lost updates, 385
- LOWER, 75
- LTRIM, 76

M

- many-to-many relationships, 439, 441
- massively parallel processing (MPP) architecture, 11–13
- master databases, 16
- MATCH, 438–450
 - FROM and, 444–448
 - AND in, 446–448
 - edge constraints in, 442–444
 - EXISTS and, 449–450
 - joins and, 441–442
 - MERGE and, 477–480
 - in SELECT queries, 438–441
 - simple match patterns, 454–455
 - UNION and, 448–449
 - WHERE and, 446–448
- matching predicates, 129–130
- MAX, 36, 66, 456
- MAXRECURSION, 188
- MAXVAL, 303
- MAXVALUE, 302–303
- .mdf file extension, 19

MERGE

MERGE

- INTO clause, 478
 - ON clause, 478
 - data manipulation language (DML)
 - triggers, 518–519
 - exercise descriptions, 333–337
 - exercise solutions, 337–341
 - OUTPUT and, 330–331
 - overview of, 317–321
 - SQL Graph and, 477–480
 - table expressions, modifying data through, 321–324
 - temporal table modification, 348–353
 - USING clause, 478
- MERGE INTO, 319
- merging of data, SQL Graph and, 477–480. *See also* MERGE
- metadata, querying of, 103–106
 - catalog views, 104–105
 - information schema views, 105
 - in SQL Graph, 434–438
 - system stored procedures and functions, 105–106
- Microsoft SQL Server. *See* SQL Server
- MIN, 36, 456
- MINVAL, 303
- MINVALUE, 302–303
- missing values
 - in outer joins, 130–132
 - overview of, 7–8
- model databases, 16–17
- modeling, graph. *See* graph modeling
- modeling, traditional, 411–417
- MONTH, 98–99
- msdb database, 17
- MTD calculations, 243
- multi-join queries
 - exercise descriptions, 137–143
 - exercise solutions, 143–147
 - outer joins in, 133–135
 - use and syntax, 127
- multiplication operator (*), 50–53
- multiset theory, 40
- multi-statement table-valued functions (TVFs), 196
- multivalued subqueries, 151–155
 - named instances, 15
 - node, 417–418
 - table, 411
- natural joins, 125
- NCHAR, 51, 64–66
- .ndf file extension, 19
- nested DML, 331–333
- nested queries
 - definition of, 149
 - derived tables and, 181–182
- NEWID, 514
- NEXT VALUE FOR, 303–307, 327
- next values, returning, 159–160
- NO ACTION, 423
- NO CACHE, 303
- NO CYCLE, 303
- NOCOUNT, 308
- node tables, creating, 420–422
- NODE_ID_FROM_PARTS, 437–438
- nodes, definition of, 417
- NOLOCK, 381
- non-equi joins, 125–127
- nonrepeatable reads, 384
- nonstandard operators, 50–53
- normal forms, 8–11
- normalization, 8–11
- NOT, 51
 - MATCH and, 448–450
 - self-contained multivalued subqueries and, 153
 - subqueries, NULL trouble and, 161–163
- NOT EXISTS, 219
 - correlated subqueries and, 172
 - MATCH and, 450
 - subqueries, NULL trouble and, 163
- NOT NULL, 130
 - GROUPING and GROUPING_ID functions and, 260–262
 - subqueries, NULL trouble and, 163
- NTILE, 234–237
- NULL, 160–161. *See also* constraints
 - aggregate functions and, 35
 - CASE expressions and, 53–56
 - COALESCE and, 70
 - data integrity, defining, 23–25
 - distinct predicate and, 211–212
 - gap filling and, 278
 - GREATEST/LEAST and, 62
 - GROUPING function and, 260–262
 - identity property and, 300
 - IF . . . ELSE flow element and, 497–498

- inner joins and, 122
 - INSERT VALUES and, 294
 - MATCH and, 451
 - missing values and, 7–8
 - missing values, overview of, 7
 - outer joins and, 145–146
 - COUNT aggregate and, 136
 - description of, 128–130
 - filtering attributes, 132–133
 - multi-join queries, 133–135
 - OUTPUT and, 331
 - overview of, 56–61
 - pivoting data, 248–249
 - string concatenation and, 69–70
 - STUFF and, 75
 - subqueries and
 - correlated subqueries, 159–160
 - NULL trouble, 161–163, 173
 - scalar subqueries, 151
 - self-contained multivalued subqueries, 153
 - table creation and, 21–22
 - table expressions, modifying data through, 322
 - unpivoting data
 - APPLY, 255
 - overview of, 252
 - UNPIVOT, 255–256
 - variables and, 493
 - window functions and, 237–242
- NUMERIC, 52, 157
- NVARCHAR, 51, 64–66, 81
- NVARCHAR(MAX), 81
- ## O
- OBJECT_DEFINITION, 192–193
- OBJECT_ID_FROM_EDGE_ID, 437–438
- OBJECT_ID_FROM_NODE_ID, 437
- OBJECT_NAME, 376
- OBJECTPROPERTY, 106
- objects
 - sequence, 302–307
 - in set theory, 4
 - SQL Server, 19–20
- OFFSET, 191
- offset window functions, 237–242
- OFFSET-FETCH, 191–192
 - APPLY operator and, 199
 - data modification with, 324–326
 - derived tables and, 178
 - exercise descriptions, 333–337
 - exercise solutions, 337–341
 - overview of, 47–48
 - views and ORDER BY clause, 222
- OLTP (online transactional processing), 11–12
- ON
 - inner joins and, 122, 123
 - MERGE and, 478
 - non-equi joins and, 125
 - outer joins and, 128, 129–130, 146
- ON DELETE CASCADE, 25
- ON DELETE, 25
- ON UPDATE, 25
- one-to-many relationships, 439
- online transactional processing (OLTP), 11–12
- ONLY, OFFSET-FETCH and, 48
- on-premises RDBMS, 13–14
- operators. *See also* APPLY; JOIN; PIVOT; set operators; UNPIVOT
 - APPLY, 63, 253–255
 - exercise descriptions, 201–206
 - exercise solutions, 206–209
 - arithmetic, 50–53
 - comparison, 50–53
 - logical, 50–53
 - overview of, 50–53, 68–69
 - precedence of, 50–53, 220–221
 - set
 - EXCEPT, 217–220
 - exercise descriptions, 223–226
 - exercise solutions, 227–230
 - INTERSECT, 214–217
 - overview of, 211–212
 - precedence and, 220–221
 - UNION, 212–214
 - unsupported logical phrases, circumventing, 221–223
- optimistic concurrency, 372
- optimization, query, 311
- OR, 51, 448–450
- ORDER BY
 - CASE expressions, 53–56
 - circumventing supported logical phrases with, 221–223
 - cursors, 500–505
 - INTERSECT ALL and, 215–217
 - OFFSET-FETCH and, 324
 - overview of, 42–44
 - ROW_NUMBER and, 234
 - set operators and, 211
 - with table expressions

ORDER BY

ORDER BY (*continued*)
 derived tables, 178
 views, 190–192
TOP filter and, 325
WINDOW clause, 244–246
window functions and, 232–233, 238
ORIGINAL_LOGIN, 518
OSQL, 494
OUTER APPLY, 197–200
outer joins, 219
 COUNT aggregate and, 136–137
 description of, 128–130
 exercise descriptions, 137–143
 exercise solutions, 143–147
 filtering attributes from nonpreserved
 side of, 132–133
 MATCH and, 438–450
 missing values, inclusion of, 130–132
 in multi-join queries, 133–135
outer queries
 definition of, 149
 in table expressions, 177
OUTPUT
 INTO and, 326
 DELETE and, 328–329
 exercise descriptions, 333–337
 exercise solutions, 337–341
 INSERT and, 326–327
 MERGE and, 330–331
 nested DML, 331–333
 overview of, 326
 SQL Graph and, 476–477
 stored procedures and, 516–517
 UPDATE and, 329–330
OVER
 sequence objects and, 305
 window functions and, 231–233, 242–244

P

PaaS (platform as a service), 14
parentheses
 in grouping sets, 258
 set operators and, 221
PARSE, 87–88, 91–93
parsing, batches and, 494–495
PARTITION BY
 INTERSECT ALL and, 215–217
 window functions and, 49–50, 233
partitions, 309–310, 373

PATINDEX, 72
PERCENT
 OFFSET-FETCH and, 48
 TOP and, 45
percent (%) wildcard, 81–82
PERIOD FOR SYSTEM_TIME, temporal table creation,
344–348, 363
pessimistic concurrency, 372
phantoms and phantom reads, 386–387
physical query processing, 117
PIVOT, 249–251, 512–513
pivoting data
 exercise descriptions, 280–285
 exercise solutions, 285–291
 with grouped queries, 248–249
 overview of, 285–291
 with PIVOT operator, 249–251
 unpivoting data
 with APPLY operator, 253–255
 overview of, 251–252
platform as a service (PaaS), 14
plus sign (+), 457
 addition operator, 50–53
 concatenation operator, 69–70
policies
 enforcement of
 with data definition language (DDL) triggers,
 519–521
 with data manipulation language (DML)
 triggers, 518–519
 retention, 344, 353
posts, in traditional modeling, 412
pound sign (#), 507
precedence
 data types, 52, 85
 operators, 50–53, 220–221
predicates. *See also specific predicate names*
 conjunction of, 50
 disjunction of, 50
 filtering, 129–130
 matching, 129–130
 overview of, 50–53
 predicate logic, 4–5
 relational model, overview of, 5–7
preserved tables, 128
previous values, returning, 159–160
primary key constraints, 9, 23
PRINT, 495, 497
 error handling, 522–525
 EXEC command and, 511

- private cloud, 14
- procedural data integrity, 22
- procedures, system, 105–106
- programmable objects
 - batches and, 494–497
 - cursors, 500–505
 - dynamic SQL, 510–513
 - error handling, 521–525
 - flow elements, 497–500
 - routines
 - overview of, 513
 - stored procedures, 511–512, 515–517
 - triggers, 517–521
 - user-defined functions (UDFs), 514–515
 - temporary tables, 505–510
 - variables, 491–493
- propositions, relational model, 5–7
- publications, in traditional modeling, 412

Q

- queries. *See also* SELECT; subqueries; table expressions; window functions
 - all-at-once operations, 63–64
 - CASE expressions, overview of, 53–56
 - character data, working with
 - CHARINDEX, 72
 - collation, 66–68
 - COMPRESS/DECOMPRESS, 79
 - CONCAT, 69–70
 - CONCAT_WS, 69–70
 - concatenation, 69–70
 - data types, 64–66
 - DATALENGTH, 71–72
 - FORMAT, 78
 - LEFT, 71
 - LEN, 71–72
 - LIKE, 81–83
 - LOWER, 75
 - PATINDEX, 72
 - REPLICATE, 72–73, 74–75
 - RIGHT, 71
 - RTRIM and LTRIM, 76
 - STRING_AGG, 81
 - STRING_SPLIT, 80
 - STUFF, 75
 - SUBSTRING, 70–71
 - TRANSLATE, 73–74, 77
 - TRIM, 76–78
 - UPPER, 75
 - FROM clause, overview of, 29–30
 - clauses versus phrases in, 29
 - cursors returned by, 43
 - date and time data, working with
 - AT TIME ZONE, 94–96
 - CAST, 91–93
 - CONVERT, 91–93
 - current date and time functions, 90–91
 - CURRENT_TIMESTAMP, 90–91
 - data ranges, filtering, 90
 - data types, 84
 - DATEADD, 96
 - DATEDIFF, 96–98
 - DATEDIFF_BIG, 96–98
 - DATENAME, 99
 - DATEPART, 98
 - DATETRUNC, 99–100
 - EOMONTH, 101–102
 - FROMPARTS, 101
 - GENERATE_SERIES, 102–103
 - GETDATE, 90–91
 - GETUTCDATE, 90–91
 - ISDATE, 100
 - literals, 84–88
 - PARSE, 91–93
 - SWITCHOFFSET, 93–94
 - SYSDATETIME, 90–91
 - SYSDATETIMEOFFSET, 90–91
 - SYSUTCDATETIME, 90–91
 - TODATETIMEOFFSET, 94
 - TRY_CAST, 91–93
 - TRY_CONVERT, 91–93
 - TRY_PARSE, 91–93
 - working with separately, 88–90
 - YEAR, MONTH, and DAY, 98–99
 - exercise descriptions, 107–111
 - exercise solutions, 111–116
 - graph data
 - data modification considerations, 474–480
 - features that are still missing, 471–473
 - LAST_NODE, 464–471
 - MATCH, 438–450, 477–480
 - overview of, 438
 - recursive queries, 450–454
 - SHORTEST_PATH, 454–464
 - GREATEST function, overview of, 62–63
 - GROUP BY clause, overview of, 32–36
 - identifier names, delimiting, 31
 - inner, 178–179
 - intersecting, 214–217

expressions; window functions

expressions; window functions (*continued*)

- LEAST function, overview of, 62–63
 - logical query processing, 117
 - metadata, querying of, 103–106
 - catalog views, 104–105
 - information schema views, 105
 - in SQL Graph, 434–438
 - system stored procedures and functions, 105–106
 - nested
 - definition of, 149
 - derived tables and, 181–182
 - NULLs, overview of, 56–61
 - OFFSET-FETCH filter, overview of, 47–48
 - operators, overview of, 50–53
 - optimization, 311
 - optimization of, 311
 - ORDER BY clause, overview of, 42–44
 - outer, 149, 177
 - physical query processing, 117
 - pivoting data with, 248–249
 - predicates, overview of, 50–53
 - recursive, 450–452, 453–454
 - SELECT clause, overview of, 37–42
 - set difference operations, 217–220
 - in temporal tables, 353–359
 - TOP filter, overview of, 44–47
 - unifying results of, 212–214
 - WHERE clause, overview of, 31–32
- QUOTED_IDENTIFIER, 68
- QUOTENAME, 513

R

- RAND, 514
- RANGE, 233
- RANK, 234–237
- ranking window functions, 234–237
- RDBMS (relational database management systems)
 - cloud, 14–15
 - definition of, 1
 - on-premises, 13–14
 - relational model, 5–7
- READ COMMITTED
 - isolation, overview of, 382–384
 - SNAPSHOT and, 371–372, 392–393
 - troubleshooting blocking and, 374
- READ COMMITTED SNAPSHOT
 - isolation levels and, 381, 392–393
 - lock modes and compatibility, 371–372
 - overview of, 371
- READ UNCOMMITTED, 381–382
- READCOMMITTEDLOCK, 392
- recursive common table expressions (CTEs), 186–188
- recursive queries, 450–454
 - overview of, 450–452
 - sorting and indentation, 453–454
- referenced tables, 24
- referencing relation, 8
- referencing tables, 24
- relational database management systems. *See* RDBMS (relational database management systems)
- relational model
 - constraints, 8
 - data integrity, defining, 22–26
 - definition of, 5
 - language independence, 2
 - missing values, 7–8
 - normalization, 8
 - propositions, predicates, and relations, 5–7
- relationships
 - commutative, 411
 - follow, 411–412
 - friendship, 411, 468
 - indirect friendship, 455
 - many-to-many, 439, 441
 - one-to-many, 439
- REPEATABLE READ, 384–385
- REPLACE, 72–73
- REPLICATE, 74–75
- resolution, batches and, 496
- Resource database, 17
- resource types, locking of, 372–373
- RESPECT NULLS, 239–242
- RESTART WITH, 303
- retention policy, 353
- RETURN
 - inline table-valued functions (TVFs) and, 196
 - user-defined functions (UDFs) and, 514
- RETURNS TABLE, 196
- RIGHT, 71, 74–75, 128
- ROLLBACK TRAN (TRANSACTION), 518
 - isolation levels and, 382
 - lock modes and compatibility, 371
 - overview of, 367
 - troubleshooting blocking and, 374
- ROLLUP, 258–259
- routines
 - overview of, 513
 - stored procedures, 511–512, 515–517
 - triggers

- DDL, 519–521
 - definition of, 517–518
 - DML, 518–519
 - user-defined functions, 514–515
- ROW, OFFSET-FETCH and, 48
- row constructors, 316
- row versioning
 - isolation levels based on
 - overview of, 387–388
 - READ COMMITTED SNAPSHOT, 392–393
 - SNAPSHOT, 388–392
 - summary of, 394
 - overview of, 368–369
- ROW_NUMBER, 234–237, 323–324
 - EXCEPT ALL and, 219
 - INTERSECT ALL and, 215–216
 - overview of, 49–50
- rows, 234–237
- ROWS, OFFSET-FETCH and, 48
- ROWS BETWEEN, 233, 238, 244
- ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, 238
- RTRIM, 76
- rules, normalization, 8–11
- running aggregates, subqueries and, 160–161

S

- scalar subqueries, 149–151
- scalar user-defined functions (UDFs), 514–515
- SCHEMA_NAME, 104
- SCHEMABINDING, views and, 193–194
- schema-qualified object name, 20
- schemas
 - CREATE SCHEMA, 495
 - information schema views, 105
 - snowflake, 12
 - SQL Server architecture, 19–20
 - star, 12
- SCOPE_IDENTITY, 300, 326
- searched CASE expressions, 53–56
- SELECT. *See also* table expressions;
- window functions
 - aliases and, 37–42
 - all-at-once operations, 63–64
 - CASE expressions, overview of, 53–56, 248–249
 - CAST, 87–88
 - character data, working with
 - CHARINDEX, 72
 - collation, 66–68
 - CONCAT and CONCAT_WS, 69–70
 - concatenation, 69–70
 - data types, 64–66
 - DATALLENGTH, 71–72
 - LEFT, 71
 - LEN, 71–72
 - LOWER, 75
 - PATINDEX, 72
 - REPLACE, 72–73
 - REPLICATE, 74–75
 - RIGHT, 71
 - RTRIM and LTRIM, 76
 - STUFF, 75
 - SUBSTRING, 70–71
 - TRANSLATE, 73–74
 - UPPER, 75
 - AS clause and, 37
 - clauses versus phrases in, 29
 - CONVERT and, 87–88
 - correlated subqueries and, 159
 - date and time data, working with
 - AT TIME ZONE, 94–96
 - CAST, 91–93
 - CONVERT, 91–93
 - current date and time functions, 90–91
 - CURRENT_TIMESTAMP, 90–91
 - data ranges, filtering, 90
 - data types, 84
 - DATEADD, 96
 - DATEDIFF, 96–98
 - DATEDIFF_BIG, 96–98
 - DATENAME, 99
 - DATEPART, 98
 - DATETRUNC, 99–100
 - EOMONTH, 101–102
 - FROMPARTS, 101
 - GENERATE_SERIES, 102–103
 - GETDATE, 90–91
 - GETUTCDATE, 90–91
 - ISDATE, 100
 - literals, 84–88
 - PARSE, 91–93
 - SWITCHOFFSET, 93–94
 - SYSDATETIME, 90–91
 - SYSDATETIMEOFFSET, 90–91
 - SYSUTCDATETIME, 90–91
 - TODATETIMEOFFSET, 94
 - TRY_CAST, 91–93
 - TRY_CONVERT, 91–93
 - TRY_PARSE, 91–93

window functions

window functions (*continued*)

- working with separately, 88–90
 - YEAR, MONTH, and DAY, 98–99
 - DISTINCT and, 41
 - elements of, 27–29
 - exercise descriptions, 107–111
 - exercise solutions, 111–116
 - global temporary tables and, 507
 - GREATEST function, overview of, 62–63
 - GROUP BY clause, overview of, 32–36
 - HAVING clause, overview of, 36–37
 - hidden columns and, 355
 - identifier names, delimiting, 31
 - INSERT SELECT, 295–296
 - isolation levels and, 381, 392
 - LEAST function, overview of, 62–63
 - MATCH and, 438–441
 - metadata, querying of, 103–106
 - catalog views, 104–105
 - information schema views, 105
 - in SQL Graph, 434–438
 - system stored procedures and functions, 105–106
 - NULLs, overview of, 56–61
 - OFFSET-FETCH filter, overview of, 47–48
 - ORDER BY clause, overview of, 42–44
 - FROM, overview of, 29–30
 - PARSE, 87–88
 - predicates and operators, overview of, 50–53
 - processing order of, 28–29
 - ranking window functions and, 236–237
 - SELECT clause, overview of, 37–42
 - SHORTEST_PATH and, 454–464
 - stored procedures and, 516–517
 - subqueries and
 - correlated subqueries, 157
 - self-contained subqueries, 149
 - with table expressions
 - common table expressions, 183–188
 - derived tables, 177–183
 - inline table-valued functions, 196–200
 - overview of, 177
 - views, 188–195
 - temporary tables and, 505–507
 - TOP filter, overview of, 44–47
 - transactions and, 370
 - variables and, 492–493
 - WHERE clause, overview of, 31–32
 - WINDOW clause, 244–246
- SELECT *41–42
- SELECT * FROM
 - edge tables, creating, 425–434
 - identity property and, 302
 - node tables, creating, 420–422
- SELECT INTO, 297–298
- SELECT NODE_ID_FROM_PARTS, 438
- self joins
 - exercise descriptions, 137–143
 - exercise solutions, 143–147
 - overview of, 119
- self-contained subqueries
 - definition of, 149
 - exercise descriptions, 166–170
 - exercise solutions, 170–175
 - multivalued subquery examples, 151–155
 - scalar subquery examples, 149–151
- semicolon (;), 22, 29, 319
- SEQUEL (Structured English Query Language), 2–3
- sequence objects, 299, 302–307
- SERIALIZABLE
 - conflict detection and, 390–392
 - isolation level, overview of, 380–381, 386–387, 394
- SERVERPROPERTY, 106
- session IDs, 375–376
- SET
 - quoted identifiers, 68
 - UPDATE and, 340–341
 - variables and, 491–493
- SET DATEFORMAT, 84–88
- SET DEFAULT, 25
- SET LANGUAGE, 85
- SET NOCOUNT ON, 497, 516–517
- SET NULL, 25
- set operators
 - EXCEPT, 217–220
 - exercise descriptions, 223–226
 - exercise solutions, 227–230
 - INTERSECT, 214–217
 - overview of, 211–212
 - precedence and, 220–221
 - UNION, 212–214
 - unsupported logical phrases, circumventing, 221–223
- set theory, 3–4
- SET TRANSACTION ISOLATION LEVEL, 381
- sets, grouping. *See* grouping sets
- shared locks, 371–372
- SHORTEST_PATH, 454–464. *See also* LAST_NODE
- simple CASE expressions, 53–56
- simple match pattern, 454–455

- single quotes ('), 68
- single-table queries. *See* queries
- SMALLDATETIME
 - data types, 84
 - literals, 84–88
 - working with date and time separately, 88–90
- SMALLDATETIMEFROMPARTS, 101
- SNAPSHOT
 - conflict detection, 390–392
 - isolation, overview of, 388–390
 - READ COMMITTED SNAPSHOT
 - isolation levels and, 381, 392–393
 - lock modes and compatibility, 371–372
 - overview of, 371
- snowflake schema, 12
- sorting, recursive queries and, 453–454
- sp_columns, 106
- sp_executesql stored procedure, 511–512
- sp_help, 105
- sp_helpconstraint, 106
- sp_sequence_get_range, 306–307
- sp_tables, 105
- SQL (Structured Query Language), overview of, 1–2.
- See also* dynamic SQL
 - database system types, 11–13
 - history and use of, 2–3
 - relational model
 - constraints, 8
 - definition of, 5
 - language independence, 2
 - missing values, 7–8
 - normalization, 8–11
 - propositions, predicates, and relations, 5–7
 - set theory, 3–4
 - standards for use, 2–3
- SQL Graph
 - cleanup, 490
 - data modification
 - deleting of data, 474–477
 - merging of data, 477–480
 - updating of data, 474–477
 - exercise descriptions, 481–483
 - exercise solutions, 484–489
 - graph data, querying
 - features that are still missing, 471–473
 - LAST_NODE function, 464–471
 - MATCH clause, 438–450, 477–480
 - overview of, 438
 - recursive queries, 450–454
 - SHORTEST_PATH option, 454–464
 - graph modeling
 - edge tables, creating, 422–434
 - metadata, querying, 434–438
 - node tables, creating, 420–422
 - overview of, 417–419
 - LAST_NODE, 464–471
 - overview of, 409
 - sample tables for, 410
 - traditional modeling versus, 411–417
 - transitive closure, 462
- SQL Server
 - Accelerated Database Recovery (ADR), 369, 387
 - architecture overview
 - cloud computing, 14–15
 - databases, 16–19
 - instances, 15–16
 - on-premises RDBMS, 13–14
 - schemas and objects, 19–20
 - authenticated login, 17–18
 - edge constraints, 423
 - gap filling in, 276
 - In-Memory OLTP, 367
 - ledgers, 353
 - Management Studio, 375
 - online transactional processing (OLTP), 11–13
 - unique indexes, 23
- SQL Server Agent, 17
- SQL Server Integration Services (SSIS), 13
- SQL Server Management Studio (SSMS), 494
 - pivoting data and, 249
 - session ID, troubleshooting blocks, 373–380
 - temporal table creation, 345
- SQL_VARIANT, 307, 507
- SQLCMD, 494
- square brackets ([]), 68, 82
- star schema, 12
- START WITH, 303
- stored procedures
 - ENCRYPTION, 192–193
 - overview of, 515–517
 - sp_columns, 106
 - sp_executesql, 511–512
 - sp_help, 105
 - sp_helpconstraint, 106
 - sp_sequence_get_range, 306–307
 - sp_tables, 105
- STRING_AGG, 81, 456, 458, 513
- STRING_SPLIT, 80
- strings, operations on. *See* character data, working with
- Structured English Query Language (SEQUEL), 2–3

STUFF

- STUFF, 75
 - subqueries. *See also* table expressions; window functions
 - bugs
 - NULL trouble, 160–161
 - substitution errors in subquery column names, 163–164
 - correlated, 155–157
 - exercise descriptions, 166–170
 - exercise solutions, 170–175
 - EXISTS predicate, 158–159
 - overview of, 149
 - previous or next values, returning, 159–160
 - running aggregates and, 160–161
 - self-contained
 - definition of, 149
 - multivalued subquery examples, 151–155
 - scalar subquery examples, 149–151
 - substitution errors in subquery column names, 163–164
 - SUBSTRING, 70–71
 - subtraction (-) operator, 50–53
 - SUM, 34–35, 242, 456, 472
 - surrogate keys, identity property and, 298–302
 - SWITCHOFFSET, 93–94, 95–96
 - symmetric multiprocessing (SMP) architecture, 11
 - Synapse Azure Analytics, 11–13
 - SYSDATETIME, 518
 - current date and time functions, 90–91
 - default constraints, defining, 26
 - INSERT SELECT and, 296
 - SYSDATETIMEOFFSET, 90–91
 - sys.dm_exec_connections, 376
 - sys.dm_exec_input_buffer, 377
 - sys.dm_exec_requests, 377–379
 - sys.dm_exec_sessions, 378
 - sys.dm_exec_sql_text, 377
 - sys.dm_os_waiting_tasks, 379
 - sys.dm_tran_locks, 374–376
 - system stored procedures/functions, 105–106
 - SYSTEM_TIME, temporal table queries, 353–359
 - SYSTEM_VERSIONING, temporal table creation, 344–348, 351, 353, 363
 - system-versioned temporal tables. *See* temporal tables
 - SYSUTCDATETIME, 90–91, 348, 360
- ## T
- table expressions, 39
 - APPLY operator, 197–200
 - common table expressions (CTEs)
 - arguments, use of, 185
 - column aliases, assigning, 184
 - defining multiple, 185
 - multiple references in, 186
 - overview of, 183–184
 - recursive, 186–188
 - derived tables
 - arguments, use of, 181
 - column aliases, assigning, 179–181
 - multiple references, 182–183
 - nesting, 181–182
 - overview of, 177–179
 - exercise descriptions, 201–206, 333–337
 - exercise solutions, 206–209, 337–341
 - inline table-valued functions (TVFs), 196–200
 - modifying data through, 321–324
 - overview of, 177
 - views
 - CHECK OPTION, 194–195
 - ENCRYPTION, 192–193
 - ORDER BY, 190–192
 - overview of, 188–189
 - SCHEMABINDING, 193–194
- table value constructor, 228
- tables. *See also* table expressions; temporal tables
 - creating, 19–22
 - derived
 - arguments, use of, 181
 - column aliases, assigning, 179–181
 - multiple references, 182–183
 - nesting, 181–182
 - overview of, 177–179
 - edge, 422–434
 - names, 411
 - node, 420–422
 - partitioning, 309–310
 - preserved, 128
 - referencing versus referenced, 24
 - temporary, 505–510
 - types of, 509–510
 - variables, 508–509
- table-valued functions, inline. *See* inline table-valued functions (TVFs)
- table-valued parameters (TVPs), 509
- table-valued user-defined functions (UDFs), 514–515
- tempdb database
 - local temporary tables, 505–507
 - overview of, 17
 - table variables, 508–509
- temporal tables

- creating, 344–348
- exercise descriptions, 360–362
- exercise solutions, 362–366
- modifying data in, 348–353
- overview of, 343
- querying data in, 353–359
- types of, 343
- temporary tables
 - global, 507–508
 - local, 505–507
 - overview of, 505
 - table types, 509–510
 - table variables, 508–509
- THEN, CASE expressions and, 53–56
- theta joins, 125
- TIME
 - data types, 84
 - literals, 84–88
 - working with date and time separately, 88–90
- time series data. *See also* date and time data, working with
 - DATE_BUCKET function
 - bucket logic, applying to data, 270–275
 - containing bucket, computation of start of, 268–270
 - overview of, 266–268
 - exercise descriptions, 280–285
 - exercise solutions, 285–291
 - gap filling, 275–279
 - overview of, 262–263
 - sample data for, 263–266
- TIMEFROMPARTS, 101
- to nodes, 417
- TO, TRUNCATE and, 309
- TODATETIMEOFFSET, 94
- TOP filter
 - data modification with, 324–326
 - derived tables and, 178
 - exercise descriptions, 333–337
 - exercise solutions, 337–341
 - overview of, 44–47
 - subqueries and, 170–171
 - views and ORDER BY clause, 191–192, 222
- total ordering, 46
- TRAILING, 78
- transactions
 - batches versus, 494
 - closing, 351
 - deadlocks, 394–397
 - definition of, 367
 - exercises, 397–408
 - isolation levels
 - overview of, 380–381
 - READ COMMITTED, 382–384
 - READ UNCOMMITTED, 381–382
 - REPEATABLE READ, 384–385
 - row versioning and, 387–388
 - SERIALIZABLE, 386–387
 - SNAPSHOT, 388–393
 - summary of, 394
 - locks and blocking
 - lockable resources types, 372–373
 - modes and compatibility, 371–372
 - overview of, 370–371
 - troubleshooting, 373–380
 - opening, 351–352
 - overview of, 367–370
- transitive closure, 462
- TRANSLATE, 73–74, 77
- triggers
 - DDL, 519–521
 - definition of, 517–518
 - DML, 518–519
- TRIM, 76–78
- troubleshooting
 - locks and blocking, 373–380
 - OUTPUT, 326
 - table expressions, modifying data through, 321–324
- TRUE, 50
 - all-at-once operations, 63–64
 - HAVING clause and, 36
 - IF . . . ELSE flow element and, 497
 - NULLS, overview of, 56–61
 - WHERE clause and, 32
 - WHILE flow element and, 498
- TRUNCATE, 309–310, 348
- TRY . . . CATCH
 - error handling, 521–525
 - transactions and, 370
- TRY_CAST, 91–93
- TRY_CONVERT, 91–93
- TRY_PARSE, 91–93
- T-SQL, overview of
 - coding style, 22
 - data integrity, 22–26
 - database system types, 11–13
 - language independence, 2

T-SQL, overview of

T-SQL, overview of (*continued*)

- predicate logic, 4–5
- relational model
 - constraints, 8
 - data integrity, defining, 22–26
 - definition of, 5
 - missing values, 7–8
 - normalization, 8–11
 - propositions, predicates, and relations, 5–7
- set theory, 3–4
- SQL development and, 2
- tables, creating, 19–22
- TYPE_NAME, 104–105
- types, data. *See* data types
- types, table, 509–510

U

- UNBOUNDED FOLLOWING, window functions and, 244
- UNBOUNDED PRECEDING, window functions and, 233
- underscore (_) wildcard, 82
- Unicode data types, 65, 512
- UNION
 - exercise descriptions, 223–226
 - exercise solutions, 227–230
 - MATCH and, 448–449
 - precedence and, 220–221
 - use of, 212–214
- UNION (DISTINCT), 213–214
- UNION ALL, 213, 256–257
- UNIQUE, NULL and, 61
- unique constraints, 23–24
- unique index, 23
- UNKNOWN, 50
 - HAVING clause and, 36
 - IF . . . ELSE flow element and, 497–498
 - missing values, overview of, 7
 - NULLs, overview of, 56–61
 - outer joins and
 - description of, 130
 - filtering attributes, 132–133
 - multi-join queries, 133–135
 - subqueries and
 - NULL trouble, 161–163, 173
 - scalar subqueries, 151
 - WHERE clause and, 32
 - WHILE flow element and, 498
- UNPIVOT, 63, 255–256
- unpivoting data

- with APPLY operator, 253–255
- overview of, 251–252
- with UNPIVOT operator, 63, 255–256
- unsupported logical phrases, circumventing, 221–223
- UPDATE, 316–317
 - assignment UPDATE, 316–317
 - based on a join, 314–316
 - data manipulation language (DML) triggers, 518–519
 - MERGE and, 317–321
 - nested DML, 331–333
 - OUTPUT and, 329–330
 - overview of, 311–314
 - sequence objects and, 305
 - SET and, 340–341
 - SQL Graph and, 474–477, 478
 - table expressions, modifying data through, 321–324
 - temporal table modification, 348–353
 - TOP filter in, 324–326
- UPPER, 75
- USE
 - SELECT and, 28
 - table creation, 21
- user-defined functions (UDFs), 514–515
- USING, MERGE and, 319, 478
- UTF-8 character encoding support, 65–68
- UTF-16 character encoding support, 65

V

- VALUES, 295
 - APPLY and, 253–255
 - INSERT VALUES, 293–295
 - missing, 7–8, 130–132
 - table value constructor, 228
- VARBINARY(MAX), 79
- VARCHAR, 21, 51, 64–66, 74, 81, 85, 294
- VARCHAR(MAX), 81
- variables
 - batches and, 495
 - as programmable objects, 491–493
 - table, 508–509
- vector expressions, 316
- views
 - catalog, 104–105
 - exercise descriptions, 201–206
 - exercise solutions, 206–209
 - information schema, 105
 - table expressions

CHECK OPTION, 194–195
 ENCRYPTION, 192–193
 ORDER BY, 190–192
 overview of, 188–189
 SCHEMABINDING, 193–194
 virtual machines (VM), 14

W

WHEN, CASE expressions and, 53–56
 WHEN MATCHED, 320, 478
 WHEN MATCHED AND, 321
 WHEN MATCHED THEN, 319
 WHEN NOT MATCHED, 320, 478
 WHEN NOT MATCHED BY SOURCE, 320
 WHEN NOT MATCHED THEN, 319
 WHERE
 CASE expressions, overview of, 53–56
 DELETE and, 308, 310
 graph metadata, querying, 437–438
 inner joins and, 122
 MATCH and, 446–448, 451
 outer joins and, 129–130
 exercise solutions, 143–147
 filtering attributes, 132–133
 overview of, 31–32
 SHORTEST_PATH and, 455–458
 subqueries and, 149, 170
 with table expressions, 179, 181, 182
 unpivoting data and, 255
 UPDATE and, 313, 314–316
 WINDOW clause and, 244–246
 WHILE flow element, 498–500
 whole, definition of, 3–4
 wildcards, LIKE predicate and
 [<character>-<character>]83
 ESCAPE character, 83
 [<list of characters>]82
 percent (%), 81–82
 underscore (_), 82

window functions. *See also specific function names*
 aggregate. *See also* COUNT; pivoting data
 graph path aggregate functions, 456
 NULLs and, 35
 overview of, 231–233, 242–244
 running aggregates, subqueries and, 160–161
 exercise descriptions, 280–285
 exercise solutions, 285–291
 offset, 237–242
 overview of, 49–50, 231–233
 ranking, 234–237
 subqueries and, 160
 WINDOW clause, 244–246
 window-frame clause, 233, 242–244
 window-order clause, 233, 242–244
 window-partition clause, 233, 242–244
 Windows authenticated login, 17–18
 WITH, common table expressions (CTEs) and,
 183–184
 WITH NOCHECK, 26
 WITH TIES
 OFFSET-FETCH and, 48
 ORDER BY and, 46–47
 subqueries and, 170–171
 WITHIN GROUP, 81, 456

X

XACT_ABORT, 368
 XML (Extensible Markup Language), 491
 XQuery, 520

Y-Z

YEAR, 98–99
 derived tables and, 179, 182
 SELECT clause and, 39
 WHERE clause and, 39
 YTD calculations, 243