

LEARNING MADE EASY



4th Edition

SQL

ALL-IN-ONE

for
dummies[®]
A Wiley Brand



Allen G. Taylor
with Richard Blum



SQL

ALL-IN-ONE

4th Edition

by Allen G. Taylor
with Richard Blum

for
dummies[®]
A Wiley Brand

SQL All-in-One For Dummies®, 4th Edition

Published by: **John Wiley & Sons, Inc.**, 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Copyright © 2024 by John Wiley & Sons, Inc., Hoboken, New Jersey

Media and software compilation copyright © 2024 by John Wiley & Sons, Inc. All rights reserved.

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Trademarks: Wiley, For Dummies, the Dummies Man logo, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit <https://hub.wiley.com/community/support/dummies>.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2024933440

ISBN 978-1-394-24229-0 (pbk); ISBN 978-1-394-24232-0 (ebk); ISBN 978-1-394-24231-3 (ebk)

Contents at a Glance

Introduction	1
Book 1: Getting Started with SQL	5
CHAPTER 1: Understanding Relational Databases	7
CHAPTER 2: Modeling a System	29
CHAPTER 3: Getting to Know SQL	53
CHAPTER 4: SQL and the Relational Model	65
CHAPTER 5: Knowing the Major Components of SQL	75
CHAPTER 6: Drilling Down to the SQL Nitty-Gritty	99
Book 2: Developing Relational Databases	133
CHAPTER 1: System Development Overview	135
CHAPTER 2: Building a Database Model	151
CHAPTER 3: Balancing Performance and Correctness	169
CHAPTER 4: Creating a Database with SQL	203
Book 3: Writing SQL Queries	217
CHAPTER 1: Values, Variables, Functions, and Expressions	219
CHAPTER 2: SELECT Statements and Modifying Clauses	249
CHAPTER 3: Querying Multiple Tables with Subqueries	295
CHAPTER 4: Querying Multiple Tables with Relational Operators	323
CHAPTER 5: Cursors	345
Book 4: Securing Your Data	357
CHAPTER 1: Protecting Against Hardware Failure and External Threats	359
CHAPTER 2: Protecting Against User Errors and Conflicts	391
CHAPTER 3: Assigning Access Privileges	419
CHAPTER 4: Error Handling	431
Book 5: Programming with SQL	447
CHAPTER 1: Database Development Environments	449
CHAPTER 2: Interfacing SQL to a Procedural Language	455
CHAPTER 3: Using SQL in an Application Program	461
CHAPTER 4: Designing a Sample Application	477
CHAPTER 5: Building an Application	497
CHAPTER 6: Understanding SQL's Procedural Capabilities	513
CHAPTER 7: Connecting SQL to a Remote Database	531

Book 6: Working with Advanced Data Types in SQL: XML, JSON, and PGQ	545
CHAPTER 1: Using XML with SQL	547
CHAPTER 2: Storing XML Data in SQL Tables	575
CHAPTER 3: Retrieving Data from XML Documents	599
CHAPTER 4: Using JSON with SQL	617
CHAPTER 5: Exploring Property Graph Queries	633
Book 7: Optimizing Your Database	645
CHAPTER 1: Tuning the Database	647
CHAPTER 2: Tuning the Environment	659
CHAPTER 3: Finding and Eliminating Performance Bottlenecks	681
Book 8: Appendixes	709
CHAPTER 1: SQL:2023 Reserved Words	711
CHAPTER 2: Glossary	721
Index	731

Table of Contents

INTRODUCTION	1
About This Book	1
Foolish Assumptions	2
Icons Used in This Book	3
Beyond the Book	3
Where to Go from Here	4
BOOK 1: GETTING STARTED WITH SQL	5
CHAPTER 1: Understanding Relational Databases	7
Understanding Why Today's Databases Are Better than Early Databases	8
Irreducible complexity	8
Managing data with complicated programs	8
Managing data with simple programs	11
Which type of organization is better?	11
Databases, Queries, and Database Applications	12
Making data useful	12
Retrieving the data you want — and only the data you want	12
Examining Competing Database Models	14
Looking at the historical background of the competing models	14
The hierarchical database model	15
The network database model	19
The relational database model	21
The object-oriented database model	25
The object-relational database model	25
The nonrelational NoSQL model	25
Why the Relational Model Won	26
CHAPTER 2: Modeling a System	29
Capturing the Users' Data Model	29
Identifying and interviewing stakeholders	30
Reconciling conflicting requirements	31
Obtaining stakeholder buy-in	31
Translating the Users' Data Model to a Formal Entity-Relationship Model	32
Entity-Relationship modeling techniques	33
Drawing Entity-Relationship diagrams	38
Understanding advanced ER model concepts	41
A simple example of an ER model	45
A slightly more complex example	47

	Problems with complex relationships	50
	Simplifying relationships using normalization	51
	Translating an ER model into a relational model	51
CHAPTER 3:	Getting to Know SQL	53
	Where SQL Came From	53
	Knowing What SQL Does	54
	The ISO/IEC SQL Standard	55
	Knowing What SQL Does Not Do	55
	Choosing and Using an Available DBMS Implementation	56
	Microsoft Access	57
	Microsoft SQL Server	61
	IBM DB2	61
	Oracle Database	62
	Sybase SQL Anywhere	62
	MySQL	62
	PostgreSQL	63
CHAPTER 4:	SQL and the Relational Model	65
	Sets, Relations, Multisets, and Tables	66
	Functional Dependencies	67
	Keys	68
	Views	69
	Users	70
	Privileges	70
	Schemas	71
	Catalogs	72
	Connections, Sessions, and Transactions	72
	Routines	73
	Paths	73
CHAPTER 5:	Knowing the Major Components of SQL	75
	Creating a Database with the Data Definition Language	75
	The containment hierarchy	76
	Creating tables	77
	Specifying columns	77
	Creating other objects	78
	Modifying tables	85
	Removing tables and other objects	85
	Operating on Data with the Data Manipulation Language (DML)	86
	Retrieving data from a database	86
	Adding data to a table	87
	Updating data in a table	90
	Deleting data from a table	93

Updating views doesn't make sense.	94
Maintaining Security in the Data Control Language (DCL)	95
Granting access privileges	95
Revoking access privileges.	96
Preserving database integrity with transactions.	96
CHAPTER 6: Drilling Down to the SQL Nitty-Gritty	99
Executing SQL Statements	99
Interactive SQL.	100
Challenges to combining SQL with a host language	101
Embedded SQL	101
Module language.	104
Using Reserved Words Correctly.	105
SQL's Data Types	105
Exact numerics.	106
Approximate numerics.	108
Character strings	110
Binary strings	112
Booleans	113
Datetimes	113
Intervals	115
XML type	116
ROW type	116
Collection types	118
REF types.	119
JSON types	119
User-defined types	120
Data type summary.	124
Handling Null Values.	125
Applying Constraints.	126
Column constraints.	126
Table constraints	128
Foreign key constraints.	129
Assertions	131
BOOK 2: DEVELOPING RELATIONAL DATABASES	133
CHAPTER 1: System Development Overview	135
The Components of a Database System	136
The database	136
The database engine.	137
The DBMS front end	137
The database application	137
The user	138

The System Development Life Cycle.	138
Definition phase.	139
Requirements phase.	140
Evaluation phase.	142
Design phase.	144
Implementation phase.	147
Final Documentation and Testing phase.	148
Maintenance phase.	150
CHAPTER 2: Building a Database Model	151
Finding and Listening to Interested Parties.	152
Your immediate supervisor.	152
The users.	153
The standards organization.	153
Upper management.	154
Building Consensus.	154
Gauging what people want.	156
Arriving at a consensus.	156
Building a Relational Model.	156
Reviewing the three database traditions.	157
Knowing what a relation is.	158
Functional dependencies.	158
Keys.	159
Being Aware of the Danger of Anomalies.	160
Eliminating anomalies.	161
Examining the higher normal forms.	164
The Database Integrity versus Performance Tradeoff.	166
CHAPTER 3: Balancing Performance and Correctness	169
Designing a Sample Database.	170
The ER model for Honest Abe's.	170
Converting an ER model into a relational model.	172
Normalizing a relational model.	172
Handling binary relationships.	174
A sample conversion.	179
Maintaining Integrity.	182
Entity integrity.	182
Domain integrity.	183
Referential integrity.	184
Avoiding Data Corruption.	186
Speeding Data Retrievals.	187
Hierarchical storage.	188
Full table scans.	189

Working with Indexes	189
Creating the right indexes	190
Indexes and the ANSI/ISO standard	190
Index costs	191
Query type dictates the best index.	191
Data structures used for indexes	193
Indexes, sparse and dense.	194
Index clustering	195
Composite indexes	195
Index effect on join performance	196
Table size as an indexing consideration.	196
Indexes versus full table scans	197
Reading SQL Server Execution Plans	197
Robust execution plans	197
A sample database	198
CHAPTER 4: Creating a Database with SQL.	203
First Things First: Planning Your Database	203
Building Tables.	204
Locating table rows with keys	204
Using the CREATE TABLE statement	207
Setting Constraints	209
Column constraints.	209
Table constraints	210
Keys and Indexes.	210
Ensuring Data Validity with Domains	210
Establishing Relationships between Tables.	211
Altering Table Structure	215
Deleting Tables	215
BOOK 3: WRITING SQL QUERIES.	217
CHAPTER 1: Values, Variables, Functions, and Expressions	219
Entering Data Values.	219
Row values have multiple parts.	220
Identifying values in a column.	220
Literal values don't change.	220
Variables vary.	222
Special variables hold specific values.	222
Working with Functions	223
Summarizing data with set functions.	223
Dissecting data with value functions	227
Using Expressions	237
Numeric value expressions	237
String value expressions.	238

	Datetime value expressions	238
	Interval value expressions	239
	Boolean value expressions	240
	Array value expressions	241
	Conditional value expressions	241
	Converting data types with a CAST expression	244
	Row value expressions	246
CHAPTER 2:	SELECT Statements and Modifying Clauses	249
	Finding Needles in Haystacks with the SELECT Statement	249
	Modifying Clauses	250
	FROM clauses	250
	WHERE clauses	251
	GROUP BY clauses	270
	HAVING clauses	273
	ORDER BY clauses	274
	Tuning Queries	276
	SELECT DISTINCT	277
	Temporary tables	280
	The ORDER BY clause	285
	The HAVING clause	289
	The OR logical connective	293
CHAPTER 3:	Querying Multiple Tables with Subqueries	295
	What Is a Subquery?	295
	What Subqueries Do	296
	Subqueries that return multiple values	296
	Subqueries that return a single value	298
	Quantified subqueries return a single value	301
	Correlated subqueries	304
	Using Subqueries in INSERT, DELETE, and UPDATE Statements	309
	Tuning Considerations for Statements Containing Nested Queries	312
	Tuning Correlated Subqueries	318
CHAPTER 4:	Querying Multiple Tables with Relational Operators	323
	UNION	324
	UNION ALL	326
	UNION CORRESPONDING	326
	INTERSECT	327
	EXCEPT	329
	JOINS	330

Cartesian product or cross join	330
Equi-join	332
Natural join.	334
Condition join.	335
Column-name join.	335
Inner join.	337
Outer join	337
ON versus WHERE	341
Join Conditions and Clustering Indexes	342
CHAPTER 5: Cursors	345
Declaring a Cursor.	346
The query expression	347
Ordering the query result set	347
Updating table rows	349
Sensitive versus insensitive cursors	349
Scrolling a cursor	351
Holding a cursor	351
Declaring a result set cursor	351
Opening a Cursor	352
Operating on a Single Row.	353
FETCH syntax	354
Absolute versus relative fetches	355
Deleting a row	355
Updating a row	355
Closing a Cursor.	356
BOOK 4: SECURING YOUR DATA	357
CHAPTER 1: Protecting Against Hardware Failure and External Threats.	359
What Could Possibly Go Wrong?	360
Equipment failure	360
Platform instability	362
Database design flaws	363
Data-entry errors.	363
Operator error.	363
Taking Advantage of RAID	363
Striping	364
RAID levels	366
Backing Up Your System.	368
Preparation for the worst.	368
Full or incremental backup	369
Frequency.	369
Backup maintenance	369

Coping with Internet Threats	370
Viruses	370
Trojan horses	372
Worms	373
Denial-of-service attacks	374
Ransomware	374
SQL injection attacks	374
Phishing scams	387
Zombie spambots	387
Installing Layers of Protection	388
Network-layer firewalls	388
Application-layer firewalls	388
Antivirus software	388
Vulnerabilities, exploits, and patches	388
Education	389
Alertness	389
CHAPTER 2: Protecting Against User Errors and Conflicts	391
Reducing Data-Entry Errors	392
Data types: The first line of defense	392
Constraints: The second line of defense	392
Sharp-eyed humans: The third line of defense	393
Coping with Errors in Database Design	393
Handling Programming Errors	394
Solving Concurrent-Operation Conflicts	394
Passing the ACID Test: Atomicity, Consistency, Isolation, and Durability	396
Operating with Transactions	397
Using the SET TRANSACTION statement	397
Starting a transaction	398
Committing a transaction	401
Rolling back a transaction	401
Implementing deferrable constraints	404
Getting Familiar with Locking	409
Two-phase locking	409
Granularity	410
Deadlock	410
Tuning Locks	411
Measuring performance with throughput	412
Eliminating unneeded locks	412
Shortening transactions	412
Weakening isolation levels (ver-r-ry carefully)	413
Controlling lock granularity	414
Scheduling DDL statements correctly	414

Partitioning insertions	414
Cooling hot spots.	415
Tuning the deadlock interval	415
Enforcing Serializability with Timestamps	415
Tuning the Recovery System	418
CHAPTER 3: Assigning Access Privileges	419
Working with the SQL Data Control Language	419
Identifying Authorized Users	420
Understanding user identifiers	420
Getting familiar with roles	420
Classifying Users	422
Granting Privileges	422
Looking at data	423
Deleting data	424
Adding data	424
Changing data	424
Referencing data in another table	425
Using certain database facilities	426
Responding to an event	427
Defining new data types.	427
Executing an SQL statement	427
Doing it all.	427
Passing on the power	428
Revoking Privileges	428
Granting Roles	430
Revoking Roles.	430
CHAPTER 4: Error Handling.	431
Identifying Error Conditions.	432
Getting to Know SQLSTATE	432
Handling Conditions	434
Handler declarations.	435
Handler actions and handler effects	435
Conditions that aren't handled	437
Dealing with Execution Exceptions: The WHENEVER Clause.	437
Getting More Information: The Diagnostics Area.	438
The diagnostics header area	439
The diagnostics detail area	440
Examining an Example Constraint Violation	442
Adding Constraints to an Existing Table	444
Interpreting SQLSTATE Information	444
Handling Exceptions	445

BOOK 5: PROGRAMMING WITH SQL	447
CHAPTER 1: Database Development Environments	449
Microsoft Access	449
The Jet engine.	450
DAO	450
ADO	450
ODBC	451
OLE DB	451
Files with the .mdb extension	451
The Access Database Engine	451
Microsoft SQL Server	451
IBM Db2	452
Oracle 23c	453
SQL Anywhere	453
PostgreSQL	453
MySQL	454
CHAPTER 2: Interfacing SQL to a Procedural Language	455
Building an Application with SQL and a Procedural Language	455
Access and VBA	456
SQL Server and the .NET languages	457
MySQL and C++.NET or C#	458
MySQL and C	458
MySQL and Perl	459
MySQL and Python	459
MySQL and PHP	459
MySQL and Java	459
Oracle SQL and Java	459
Db2 and Java	460
CHAPTER 3: Using SQL in an Application Program	461
Comparing SQL with Procedural Languages	462
Classic procedural languages	462
Object-oriented procedural languages	463
Nonprocedural languages	463
Difficulties in Combining SQL with a Procedural Language	464
Challenges of using SQL with a classical procedural language	464
Challenges of using SQL with an object-oriented procedural language	465
Embedding SQL in an Application	466
Embedding SQL in an Oracle Pro*C application	467
Embedding SQL in a Java application	470
Using SQL in a Perl application	470
Embedding SQL in a PHP application	471

Using SQL with a Visual Basic .NET application.	471
Using SQL with other .NET languages	472
Using SQL Modules with an Application	472
Module declarations	473
Module procedures.	474
Modules in Oracle	474
CHAPTER 4: Designing a Sample Application	477
Understanding the Client's Problem.	478
Approaching the Problem	478
Interviewing the stakeholders	478
Drafting a detailed statement of requirements	479
Following up with a proposal.	479
Determining the Deliverables	480
Finding out what's needed now and later	481
Planning for organization growth	481
Nailing down project scope	482
Building an Entity-Relationship Model	483
Determining what the entities are	484
Relating the entities to one another	484
Transforming the Model.	487
Eliminating any many-to-many relationships	487
Normalizing the ER model	490
Creating Tables	491
Changing Table Structure.	495
Removing Tables	495
Designing the User Interface	496
CHAPTER 5: Building an Application	497
Designing from the Top Down.	497
Determining what the application should include	498
Designing the user interface	498
Connecting the user interface to the database.	499
Coding from the Bottom Up	501
Preparing to build the application	501
Creating the application's building blocks	509
Gluing everything together	510
Testing, Testing, Testing	510
Fixing the bugs.	511
Turning naive users loose	511
Bringing on the hackers	511
Fixing the newly found bugs	511
Retesting everything one last time	512

CHAPTER 6:	Understanding SQL's Procedural Capabilities	513
	Embedding SQL Statements in Your Code	514
	Introducing Compound Statements	514
	Atomicity	515
	Variables	516
	Cursors	516
	Assignment	517
	Following the Flow of Control Statements	517
	IF ... THEN ... ELSE ... END IF	518
	CASE ... END CASE	518
	LOOP ... END LOOP	520
	LEAVE	520
	WHILE ... DO ... END WHILE	521
	REPEAT ... UNTIL ... END REPEAT	521
	FOR ... DO ... END FOR	522
	ITERATE	522
	Using Stored Procedures	523
	Working with Triggers	524
	Trigger events	525
	Trigger action time	526
	Triggered actions	526
	Triggered SQL statement	526
	Using Stored Functions	527
	Passing Out Privileges	528
	Using Stored Modules	528
CHAPTER 7:	Connecting SQL to a Remote Database	531
	Native Drivers	532
	ODBC and Its Major Components	533
	Application	534
	Driver manager	535
	Drivers	535
	Data sources	537
	What Happens When the Application Makes a Request	537
	Using handles to identify objects	538
	Following the six stages of an ODBC operation	538
BOOK 6: WORKING WITH ADVANCED DATA TYPES IN SQL: XML, JSON, AND PGQ		545
CHAPTER 1:	Using XML with SQL	547
	Introducing XML	548
	Knowing the Parts of an XML Document	549

XML declaration	549
Elements	550
Attributes	551
Entity references	552
Numeric character references	553
Using XML Schema	553
Relating SQL to XML	554
Using the XML Data Type	555
When to use the XML type	555
When not to use the XML type	556
Mapping SQL to XML	557
Mapping character sets to XML	557
Mapping identifiers to XML	557
Mapping data types to XML	558
Mapping nonpredefined data types to XML	559
Mapping tables to XML	564
Handling null values	564
Creating an XML schema for an SQL table	565
Operating on XML Data with SQL Functions	566
XMLELEMENT	567
XMLFOREST	567
XMLCONCAT	568
XMLAGG	568
XMLCOMMENT	569
XMLPARSE	569
XMLPI	570
XMLQUERY	570
XMLCAST	571
Working with XML Predicates	571
DOCUMENT	571
CONTENT	572
XMLEXISTS	572
VALID	572
CHAPTER 2: Storing XML Data in SQL Tables	575
Inserting XML Data into an SQL Pseudotable	575
Creating a Table to Hold XML Data	577
Updating XML Documents	578
Discovering Oracle's Tools for Updating XML Data in a Table	579
APPENDCHILDXML	579
INSERTCHILDXML	580
INSERTXMLBEFORE	581
DELETXML	582
UPDATEXML	583

Introducing Microsoft's Tools for Updating XML Data in a Table . . .	584
Inserting data into a table using OPENXML	584
Using updategrams to map data into database tables	585
Using an updategram namespace and keywords	585
Specifying a mapping schema	587
CHAPTER 3: Retrieving Data from XML Documents	599
XQuery	600
Where XQuery came from	600
What XQuery requires	601
XQuery functionality	601
Usage scenarios	602
FLWOR Expressions	607
The for clause	608
The let clause	609
The where clause	610
The order by clause	611
The return clause	611
XQuery versus SQL	612
Comparing XQuery's FLWOR expression with SQL's	
SELECT expression	613
Relating XQuery data types to SQL data types	613
CHAPTER 4: Using JSON with SQL	617
Using JSON with SQL	617
The SQL/JSON Data Model	618
SQL/JSON items	618
SQL/JSON sequences	619
Parsing JSON	620
Serializing JSON	620
SQL/JSON Functions	620
Query functions	620
Constructor functions	626
IS JSON predicate	628
JSON nulls and SQL nulls	629
SQL/JSON Path Language	629
SQL:2023 JSON Enhancements	629
The JSON data type	630
Additional functions for the JSON data type	631
CHAPTER 5: Exploring Property Graph Queries	633
What Are Property Graph Queries?	633
Looking at node and edge properties	634
Connecting nodes by multiple edges	634

Using edges to connect a node to itself	635
Following paths with SQL queries	635
Examining SQL/PGQ	635
Working with SQL/PGQ	637
Building the property graph tables	637
Adding data to node and edge tables	640
Querying data in graph tables	642
BOOK 7: OPTIMIZING YOUR DATABASE	645
CHAPTER 1: Tuning the Database	647
Analyzing the Workload	648
Considering the Physical Design	649
Choosing the Right Indexes	650
Avoiding unnecessary indexes	650
Choosing a column to index	651
Using multicolumn indexes	652
Clustering indexes	652
Choosing an index type	654
Weighing the cost of index maintenance	654
Using composite indexes	654
Tuning Indexes	655
Tuning Queries	656
Tuning Transactions	657
Separating User Interactions from Transactions	657
Minimizing Traffic between Application and Server	658
Precompiling Frequently Used Queries	658
CHAPTER 2: Tuning the Environment	659
Surviving Failures with Minimum Data Loss	660
What happens to transactions when no failure occurs?	660
What happens when a failure occurs and a transaction is still active?	661
Tuning the Recovery System	661
Volatile and nonvolatile memory	662
Memory system hierarchy	663
Putting logs and transactions on different disks	664
Tuning write operations	667
Performing database dumps	668
Setting checkpoints	669
Optimizing batch transactions	670
Tuning the Operating System	670
Scheduling threads	670
Determining database buffer size	674
Tuning the page usage factor	675

Maximizing the Hardware You Have	675
Optimizing the placement of code and data on hard disks	676
Tuning the page replacement algorithm	676
Tuning the disk controller cache	677
Adding Hardware	677
Faster processor	678
More RAM	678
Faster hard disks	678
More hard disks	679
Solid State Disk (SSD)	679
RAID arrays	679
Working in Multiprocessor Environments	680
 CHAPTER 3: Finding and Eliminating Performance Bottlenecks	681
Pinpointing the Problem	682
Slow query	682
Slow update	682
Determining the Possible Causes of Trouble	683
Problems with indexes	683
Pitfalls in communication	685
Determining whether hardware is robust enough and configured properly	687
Implementing General Principles: A First Step Toward Improving Performance	687
Avoid direct user interaction	687
Examine the application/database interaction	688
Don't ask for columns that you don't need	688
Don't use cursors unless you absolutely have to	689
Precompiled queries	689
Tracking Down Bottlenecks	689
Isolating performance problems	690
Performing a top-down analysis	690
Partitioning	692
Locating hotspots	693
Analyzing Query Efficiency	693
Using query analyzers	693
Finding problem queries	701
Managing Resources Wisely	705
The disk subsystem	705
The database buffer manager	706
The logging subsystem	707
The locking subsystem	707

BOOK 8: APPENDIXES	709
CHAPTER 1: SQL:2023 Reserved Words	711
CHAPTER 2: Glossary	721
INDEX	731

Introduction

SQL is the internationally recognized standard language for dealing with data in relational databases. Developed by IBM, SQL became an international standard in 1986. The standard was updated in 1989, 1992, 1999, 2003, 2008, 2011, 2016, and 2023. It continues to evolve and gain capability. Database vendors continually update their products to incorporate the new features of the ISO/IEC standard. (For the curious out there, ISO is the International Organization for Standardization, and IEC is the International Electrotechnical Commission.)

SQL isn't a general-purpose language, such as C++ or Java. Instead, it's strictly designed to deal with data in relational databases. With SQL, you can carry out all the following tasks:

- » Create a database, including all tables and relationships.
- » Fill database tables with data.
- » Change the data in database tables.
- » Delete data from database tables.
- » Retrieve specific information from database tables.
- » Grant and revoke access to database tables.
- » Protect database tables from corruption due to access conflicts or user mistakes.

About This Book

This book isn't just about SQL; it's also about how SQL fits into the process of creating and maintaining databases and database applications. In this book, I cover how SQL fits into the larger world of application development and how it handles data coming in from other computers, which may be on the other side of the world or even in interplanetary space.

Here are some of the things you can do with this book:

- » Create a model of a proposed system and then translate that model into a database.
- » Find out about the capabilities and limitations of SQL.
- » Discover how to develop reliable and maintainable database systems.
- » Create databases.
- » Speed database queries.
- » Protect databases from hardware failures, software bugs, and Internet attacks.
- » Control access to sensitive information.
- » Write effective database applications.
- » Deal with data from a variety of nontraditional data sources by using XML.

I've structured this book modularly — that is, it's designed so that you can easily find just the information you need — so you don't have to read whatever doesn't pertain to your task at hand. Here and there throughout the book, I include sidebars containing interesting information that isn't necessarily integral to the discussion at hand; feel free to skip them. You also don't have to read text marked with the Technical Stuff icons, which parses out über-techy tidbits (which may or may not be your cup of tea).

Within this book, you may note that some web addresses break across two lines of text. If you're reading this book in print and want to visit one of these web pages, simply key in the web address exactly as it's noted in the text, pretending as though the line break doesn't exist. If you're reading this as an e-book, you've got it easy — just click the web address to be taken directly to the web page.

Foolish Assumptions

I know that this is a *For Dummies* book, but I don't really expect that you're a dummy. In fact, I assume that you're a very smart person. After all, you decided to read this book, which is a sign of high intelligence indeed. Therefore, I assume that you may want to do a few things, such as re-create some of the examples in the book. You may even want to enter some SQL code and execute it. To do that, you need at the very least an SQL editor and more likely also a database management system (DBMS) of some sort. Many choices are available, both proprietary

and open source. I mention several of these products at various places throughout the book but don't recommend any one in particular. Any product that complies with the ISO/IEC international SQL standard should be fine.

Take claims of ISO/IEC compliance with a grain of salt, however. No DBMS available today is 100 percent compliant with the ISO/IEC SQL standard. For that reason, some of the code examples I give in this book may not work in the particular SQL implementation that you're using. The code samples I use in this book are consistent with the international standard rather than with the syntax of any particular implementation unless I specifically state that the code is for a particular implementation.

Icons Used in This Book

For Dummies books are known for those helpful icons that point you in the direction of really great information. This section briefly describes the icons used in this book.



TIP

The Tip icon points out helpful information that's likely to make your job easier.



REMEMBER

This icon marks a generally interesting and useful fact — something that you may want to remember for later use.



WARNING

The Warning icon highlights lurking danger. When you see this icon, pay attention, and proceed with caution.



TECHNICAL
STUFF

This icon denotes techie stuff nearby. If you're not feeling very techie, you can skip this info.

Beyond the Book

In addition to what you're reading right now, this book comes with a free access-anywhere Cheat Sheet that includes information on SQL system development, normalizing data, and SQL data types and functions. To get this Cheat Sheet, simply go to www.dummies.com and type **SQL All-in-One For Dummies Cheat Sheet** in the Search box.

Where to Go from Here

Book 1 is the place to go if you're just getting started with databases. It explains why databases are useful and describes the different types. It focuses on the relational model and describes SQL's structure and features.

Book 2 goes into detail on how to build a database that's reliable as well as responsive. Unreliable databases are much too easy to create, and this minibook tells you how to avoid the pitfalls that lie in wait for the unwary.

Go directly to Book 3 if your database already exists and you just want to know how to use SQL to pull from it the information you want.

Book 4 is primarily aimed at the database administrator (DBA) rather than the database application developer or user. It discusses how to build a robust database system that resists data corruption and data loss.

Book 5 is for the application developer. In addition to discussing how to write a database application, it gives an example that describes in a step-by-step manner how to build a reliable application.

If you're already an old hand at SQL and just want to know how to handle data in XML or JSON format in your SQL database, or if you'd like to dive into the property graph database world, Book 6 is for you.

Book 7 gives you a wide variety of techniques for improving the performance of your database. This minibook is the place to go if your database is operating — but not as well as you think it should. Most of these techniques are things that the DBA can do, rather than the application developer or the database user. If your database isn't performing the way you think it should, take it up with your DBA. She can do a few things that could help immensely.

Book 8 is a handy reference that helps you quickly find the meaning of a word you've encountered or see why an SQL statement that you entered didn't work as expected. (Maybe you used a reserved word without realizing it.)

1

Getting Started with SQL

Contents at a Glance

CHAPTER 1:	Understanding Relational Databases	7
	Understanding Why Today's Databases Are Better than Early Databases	8
	Databases, Queries, and Database Applications	12
	Examining Competing Database Models	14
	Why the Relational Model Won	26
CHAPTER 2:	Modeling a System	29
	Capturing the Users' Data Model	29
	Translating the Users' Data Model to a Formal Entity-Relationship Model	32
CHAPTER 3:	Getting to Know SQL	53
	Where SQL Came From	53
	Knowing What SQL Does	54
	The ISO/IEC SQL Standard	55
	Knowing What SQL Does Not Do	55
	Choosing and Using an Available DBMS Implementation	56
CHAPTER 4:	SQL and the Relational Model	65
	Sets, Relations, Multisets, and Tables	66
	Functional Dependencies	67
	Keys	68
	Views	69
	Users	70
	Privileges	70
	Schemas	71
	Catalogs	72
	Connections, Sessions, and Transactions	72
	Routines	73
	Paths	73
CHAPTER 5:	Knowing the Major Components of SQL	75
	Creating a Database with the Data Definition Language	75
	Operating on Data with the Data Manipulation Language (DML) ...	86
	Maintaining Security in the Data Control Language (DCL)	95
CHAPTER 6:	Drilling Down to the SQL Nitty-Gritty	99
	Executing SQL Statements	99
	Using Reserved Words Correctly	105
	SQL's Data Types	105
	Handling Null Values	125
	Applying Constraints	126

- » Working with data files and databases
- » Seeing how databases, queries, and database applications fit together
- » Looking at different database models
- » Charting the rise of relational databases

Chapter **1**

Understanding Relational Databases

SQL (pronounced *ess cue el*, but you'll hear some people say *see quel*) is the international standard language used in conjunction with relational databases — and it just so happens that relational databases are the dominant form of data storage throughout the world. In order to understand *why* relational databases are the primary repositories for the data of both small and large organizations, you must first understand the various ways in which computer data can be stored and how those storage methods relate to the relational database model. To help you gain that understanding, I spend a good portion of this chapter going back to the earliest days of electronic computers and recapping the history of data storage.

I realize that grand historical overviews aren't everybody's cup of tea, but I'd argue that it's important to see that the different data storage strategies that have been used over the years each have their own strengths and weaknesses. Ultimately, the strengths of the relational model overshadowed its weaknesses and it became the most frequently used method of data storage. Shortly after that, SQL became the most frequently used method of dealing with data stored in a relational database.

Understanding Why Today's Databases Are Better than Early Databases

In the early days of computers, the concept of a database was more theoretical than practical. Vannevar Bush, the 20th-century visionary, conceived of the idea of a database in 1945, even before the first electronic computer was built. However, practical implementations of databases — such as IBM's IMS (Information Management System), which kept track of all the parts on the Apollo moon mission and its commercial followers — did not appear for a number of years after that. For far too long, computer data was still being kept in files rather than migrated to databases.

Irreducible complexity

Any software system that performs a useful function is complex. The more valuable the function, the more complex its implementation. Regardless of how the data is stored, the complexity remains. The only question is where that complexity resides.

Any nontrivial computer application has two major components: the program and the data. Although an application's level of complexity depends on the task to be performed, developers have some control over the location of that complexity. The complexity may reside primarily in the program part of the overall system, or it may reside in the data part. In the sections that follow, I tell you how the location of complexity in databases shifted over the years as technological improvements made that possible.

Managing data with complicated programs

In the earliest applications of computers to solve problems, all of the complexity resided in the program. The data consisted of one data record of fixed length after another, stored sequentially in a file. This is called a *flat file* data structure. The data file contains nothing but data. The program file must include information about where particular records are within the data file (one form of *metadata*, whose sole purpose is to organize the primary data you *really* care about). Thus, for this type of organization, the complexity of managing the data is entirely in the program.

Here's an example of data organized in a flat file structure:

```
Harold Percival 26262 S. Howards Mill Rd.Westminster CA92683
Jerry Appel 32323 S. River Lane Road Santa Ana CA92705
Adrian Hansen 232 Glenwood Court Anaheim CA92640
John Baker 2222 Lafayette Street Garden GroveCA92643
Michael Pens 77730 S. New Era Road Irvine CA92715
Bob Michimoto 25252 S. Kelmsley Drive Stanton CA92610
Linda Smith 444 S.E. Seventh StreetCosta Mesa CA92635
Robert Funnell 2424 Sheri Court Anaheim CA92640
Bill Checkal 9595 Curry Drive Stanton CA92610
Jed Style 3535 Randall Street Santa Ana CA92705
```

This example includes fields for name, address, city, state, and zip code. Each field has a specific length, and data entries must be truncated to fit into that length. If entries don't use all the space allotted to them, storage space is wasted.

The flat file method of storing data has several consequences, some beneficial and some not. First, the beneficial consequences:

- » **Storage requirements are minimized.** Because the data files contain nothing but data, they take up a minimum amount of space on hard disks or other storage media. The code that must be added to any one program that contains the metadata is small compared to the overhead involved with adding a database management system (DBMS) to the data side of the system. (A *database management system* is the program that controls access to — and operations on — a database.)
- » **Operations on the data can be fast.** Because the program interacts directly with the data, with no DBMS in the middle, well-designed applications can run as fast as the hardware permits.

Wow! What could be better? A data organization that minimizes storage requirements and at the same time maximizes speed of operation seems like the best of all possible worlds. But wait a minute . . .

Flat file systems came into use in the 1940s. We have known about them for a long time, and yet today they are almost entirely replaced by database systems. What's up with that? Perhaps it is the not-so-beneficial consequences:

- » **Updating the data's structure can be a huge task.** It is common for an organization's data to be operated on by multiple application programs, with multiple purposes. If the metadata about the structure of data is in the program rather than attached to the data itself, *all* the programs that access

that data must be modified whenever the data structure is changed. Not only does this cause a lot of redundant work (because the same changes must be made in all the programs), but it is an invitation to problems. All the programs must be modified in exactly the same way. If one program is inadvertently forgotten, the program will fail the next time you run it. Even if all the programs *are* modified, any that aren't modified exactly as they should be will fail, or even worse, corrupt the data without giving any indication that something is wrong.

- » **Flat file systems provide no protection of individual data elements.** With flat files, you have read/write access either to the entire file or to none of the file. A flat file system doesn't have a database management system, which allows you to restrict types of access to the data to only authorized users.
- » **Speed can be compromised.** Accessing records in a large flat file can actually be slower than a similar access in a database because flat file systems do not support indexing. Indexing is a major topic that I discuss in Book 2, Chapter 3.
- » **Portability becomes an issue.** If the specifics that handle how you retrieve a particular piece of data from a particular disk drive is coded into each program, what happens when your hardware becomes obsolete and you must migrate to a new system? All your applications will have to be changed to reflect the new way of accessing the data. This task is so onerous that many organizations have chosen to limp by on old, poorly performing systems instead of enduring the pain of transitioning to a system that would meet their needs much more effectively. Organizations with legacy systems consisting of millions of lines of code are pretty much trapped.

In the early days of electronic computers, storage was relatively expensive, so system designers were highly motivated to accomplish their tasks using as little storage space as possible. Also, in those early days, computers were much slower than they are today, so doing things the fastest possible way also had a high priority. Both of these considerations made flat file systems the architecture of choice, despite the problems inherent in updating the structure of a system's data.

The situation today is radically different. The cost of storage has plummeted and continues to drop on an exponential curve. The speed at which computations are performed has increased exponentially also. As a result, minimizing storage requirements and maximizing the speed with which an operation can be performed are no longer the primary driving forces that they once were. Because systems have continually become bigger and more complex, the problem of maintaining them has likewise grown. For all these reasons, flat file systems have lost their attractiveness, and databases have replaced them in practically all application areas.

Managing data with simple programs

The major selling point of database systems is that the metadata resides on the data end of the system rather than in the program. The program doesn't have to know anything about the details of how the data is stored. The program makes *logical* requests for data, and the DBMS translates those logical requests into commands that go out to the physical storage hardware to perform whatever operation has been requested. (In this context, a *logical request* asks for a specific piece of information, but does not specify its location on hard disk in terms of platter, track, sector, and byte.) Here are the advantages of this organization:

- » Because application programs need to know only what data they want to operate on, and not where that data is located, they are unaffected when the physical details of where data is stored changes.
- » Portability across platforms, even when they are highly dissimilar, is easy as long as the DBMS used by the first platform is also available on the second. Generally, you don't need to change the programs at all to accommodate various platforms.

What about the disadvantages? They include the following:

- » Placing a database management system in between the application program and the data slows down operations on that data. This is not nearly the problem that it used to be. Modern advances, such as the use of high speed cache memories have eased this problem considerably.
- » Databases take up more space on disk storage than the same amount of data would take up in a flat file system. This is due to the fact that metadata is stored along with the data. The metadata contains information about how the data is stored so that the application programs don't have to include it.

Which type of organization is better?

I bet you think you already know how I'm going to answer this question. You're probably right, but the answer is not quite so simple. There is no one correct answer that applies to all situations. In the early days of electronic computing, flat file systems were the only viable option. To perform any reasonable computation in a timely and economical manner, you had to use whatever approach was the fastest and required the least amount of storage space. As more and more application software was developed for these systems, the organizations that owned them became locked in tighter and tighter to what they had. To change to a more modern database system requires rewriting all their applications from scratch and reorganizing all their data, a monumental task. As a result, we still have legacy

flat file systems that continue to exist because switching to more modern technology isn't feasible, both economically and in terms of the time it would take to make the transition.

Databases, Queries, and Database Applications

What are the chances that a person could actually find a needle in a haystack? Not very good. Finding the proverbial needle is so hard because the haystack is a random pile of hay with individual pieces of hay going in every direction, and the needle is located at some random place among all that hay.

A flat file system is not really very much like a haystack, but it does lack structure — and in order to find a particular record in such a file, you must use tools that lie outside of the file itself. This is like applying a powerful magnet to the haystack to find the needle.

Making data useful

For a collection of data to be useful, you must be able to easily and quickly retrieve the particular data you want, without having to wade through all the rest of the data. One way to make this happen is to store the data in a logical structure. Flat files don't have much structure, but databases do. Historically, the hierarchical database model and the network database model were developed before the relational model. Each one organizes data in a different way, but all three produce a highly structured result. Because of that, starting in the 1970s, any new development projects were most likely done using one of the aforementioned three database models: hierarchical, network, or relational. (I explore each of these database models further in the “Examining Competing Database Models” section, later in this chapter.)

Retrieving the data you want — and only the data you want

Of all the operations that people perform on a collection of data, the retrieval of specific elements out of the collection is the most important. This is because retrievals are performed more often than any other operation. Data entry is done only once. Changes to existing data are made relatively infrequently, and data is

deleted only once. Retrievals, on the other hand, are performed frequently, and the same data elements may be retrieved many times. Thus, if you could optimize only one operation performed on a collection of data, that one operation should be data retrieval. As a result, modern database management systems put a great deal of effort into making retrievals fast.

Retrievals are performed by queries. A modern database management system analyzes a query that is presented to it and decides how best to perform it. Generally, there are multiple ways of performing a query, some much faster than others. A good DBMS consistently chooses a near-optimal execution plan. Of course, it helps if the query is formulated in an optimal manner to begin with. (I discuss optimization strategies in depth in Book 7, which covers database tuning.)

THE FIRST DATABASE SYSTEM

The first true database system was developed by IBM in the 1960s in support of NASA's Apollo moon landing program. The number of components in the Saturn V launch vehicle, the Apollo Command and Service Module, and the lunar lander far exceeded anything that had been built up to that time. Every component had to be tested more exhaustively than anything had ever been tested before because each component would have to withstand the rigors of an environment that was more hostile and more unforgiving than any environment that humans had ever attempted to work in. Flat file systems were out of the question. IBM's solution, which IBM later transformed into a commercial database product named IMS (Information Management System), kept track of each individual component, as well as its complete history.

When the ill-fated Apollo 13's main oxygen tank ruptured on the way to the Moon, engineers worked frantically to come up with a plan to save the lives of the three astronauts headed for the Moon. The engineers succeeded and transmitted a plan to the astronauts that worked.

After the crew had returned safely to Earth, querying IMS records about the oxygen tank that failed showed that somewhere between the oxygen tank's manufacture and its installation in Apollo 13, it had been dropped on the floor. Engineers retested it for its ability to withstand the pressure it would have to contain during the mission, and then put it back in stock after it passed the test. But it turns out that in this case, the test did not detect the hidden damage to the tank, and NASA should not have used the oxygen tank on the Apollo 13 mission. The history stored in IMS showed that passing a pressure test is not enough to assure that a dropped tank is undamaged. No dropped tanks were ever used on subsequent Apollo missions.

Examining Competing Database Models

A *database model* is simply a way of organizing data elements within a database. In this section, I give you the details on the three database models that appeared first on the scene:

- » **Hierarchical:** Organizes data into levels, where each level contains a single category of data, and parent/child relationships are established between levels
- » **Network:** Organizes data in a way that avoids much of the redundancy inherent in the hierarchical model
- » **Relational:** Organizes data into a structured collection of two-dimensional tables

After the introductions of the hierarchical, network, and relational models, computer scientists have continued to develop databases models that have been found useful in some categories of applications. I briefly mention some of these later in this chapter, along with their areas of applicability. However, the hierarchical, network, and relational models are the ones that have been primarily used for general business applications.

Looking at the historical background of the competing models

The first functioning database system was developed by IBM and went live at an Apollo contractor's site on August 14, 1968. (Read the whole story in "The first database system" sidebar, here in this chapter.) Known as IMS (Information Management System), it is still (amazingly enough) in use today, over 50 years later, because IBM has continually upgraded it in support of its customers.



TIP

If you are in the market for a database management system, you may want to consider buying it from a vendor that will be around, and that is committed to supporting it for as long as you will want to use it. IBM has shown itself to be such a vendor, and of course, there are others as well.

IMS is an example of a hierarchical database product. About a year after IMS was first run, the network database model was described by an industry committee. About a year after that, Dr. Edgar F. “Ted” Codd, also of IBM, proposed the relational model. Within a short span of years, the three models that were to dominate the database market for decades were spawned.

Quite a few years went by before the object-oriented database model made its appearance, presenting itself as an alternative meant to address some of the deficiencies of the relational model. The *object-oriented database model* accommodates the storage of types of data that don’t easily fit into the categories handled by relational databases. Although they have advantages in some applications, object-oriented databases have not captured significant market share. The *object-relational model* is a merger of the relational and object models, and it is designed to capture the strengths of both, while leaving behind their major weaknesses. Now, there is something called the NoSQL model, which stores data as documents instead of tables. The most popular NoSQL model is the MongoDB database system. Because NoSQL stores data as documents, it is designed mostly to work with data that is not rigidly structured. Because it does not use SQL, I will not discuss it in this book.

The hierarchical database model

The *hierarchical database model* organizes data into levels, where each level contains a single category of data, and parent/child relationships are established between levels. Each parent item can have multiple children, but each child item can have one and only one parent. Mathematicians call this a *tree-structured* organization, because the relationships are organized like a tree with a trunk that branches out into limbs that branch out into smaller limbs. Thus all relationships in a hierarchical database are either one-to-one or one-to-many. Many-to-many relationships are not used. (More on these kinds of relationships in a bit.)

A list of all the stuff that goes into building a finished product — a listing known as a *bill of materials*, or BOM — is well suited for a hierarchical database. For example, an entire machine is composed of assemblies, which are each composed of subassemblies, and so on, down to individual components. As an example of such an application, consider the mighty Saturn V Moon rocket that sent American astronauts to the Moon in the late 1960s and early 1970s. Figure 1-1 shows a hierarchical diagram of major components of the Saturn V.

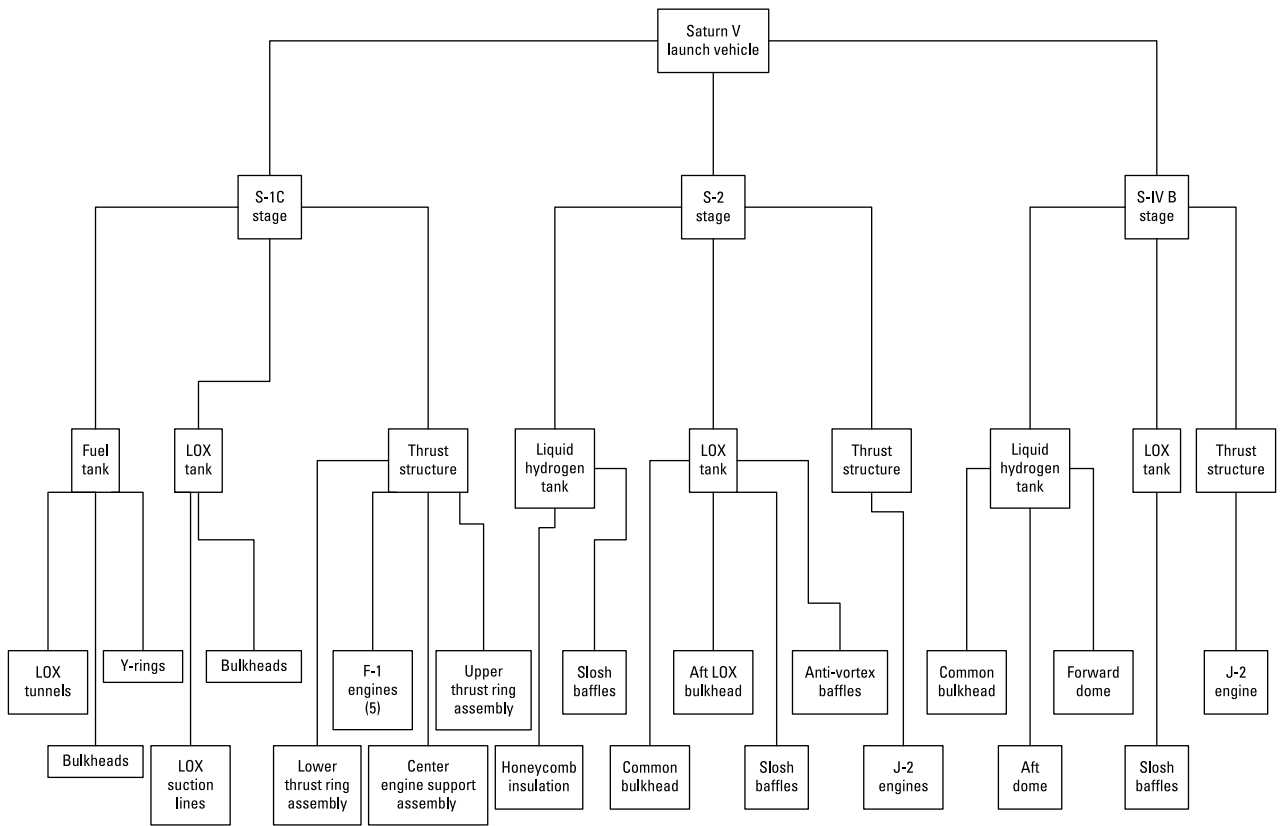


FIGURE 1-1:
A hierarchical
model of the
Saturn V moon
rocket.

Three relationships can occur between objects in a database:

- » **One-to-one relationship:** One object of the first type is related to one and only one object of the second type. In Figure 1-1, there are several examples of one-to-one relationships. One is the relationship between the S-2 stage LOX tank and the aft LOX bulkhead. Each LOX tank has one and only one aft LOX bulkhead, and each aft LOX bulkhead belongs to one and only one LOX tank.
- » **One-to-many relationship:** One object of the first type is related to multiple objects of the second type. In the Saturn V's S-1C stage, the thrust structure contains five F-1 engines, but each engine belongs to one and only one thrust structure.
- » **Many-to-many relationship:** Multiple objects of the first type are related to multiple objects of the second type. This kind of relationship is not handled cleanly by a hierarchical database. Attempts to do so tend to be kludgy. One example might be two-inch hex-head bolts. These bolts are not considered to be uniquely identifiable, and any one such bolt is interchangeable with any other. An assembly might use multiple bolts, and a bolt could be used in any of several different assemblies.

A great strength of the hierarchical model is its high performance. Because relationships between entities are simple and direct, retrievals from a hierarchical database that are set up to take advantage of the way the data is structured can be very fast. However, retrievals that don't take advantage of the way the data is structured are slow and sometimes can't be made at all. It's difficult to change the structure of a hierarchical database to address new requirements. This structural rigidity is the greatest weakness of the hierarchical model. Another problem with the hierarchical model is the fact that, structurally, it requires a lot of redundancy, as my next example makes clear.

First off, time to state the obvious: Not many organizations today are designing rockets capable of launching payloads to the moon. The hierarchical model can also be applied to more common tasks, however, such as tracking sales transactions for a retail business. As an example, I use some sales transaction data from Gentoo Joyce's fictitious online store of penguin collectibles. She accepts PayPal, MasterCard, Visa, and money orders and sells various items featuring depictions of penguins of specific types — gentoo, chinstrap, and adolie.

As shown in Figure 1-2, customers who have made multiple purchases show up in the database multiple times. For example, you can see that Lynne has purchased with PayPal, MasterCard, and Visa. Because this is hierarchical, Lynne's information shows up multiple times, and so does the information for every customer who has bought more than once. Product information shows up multiple times too.

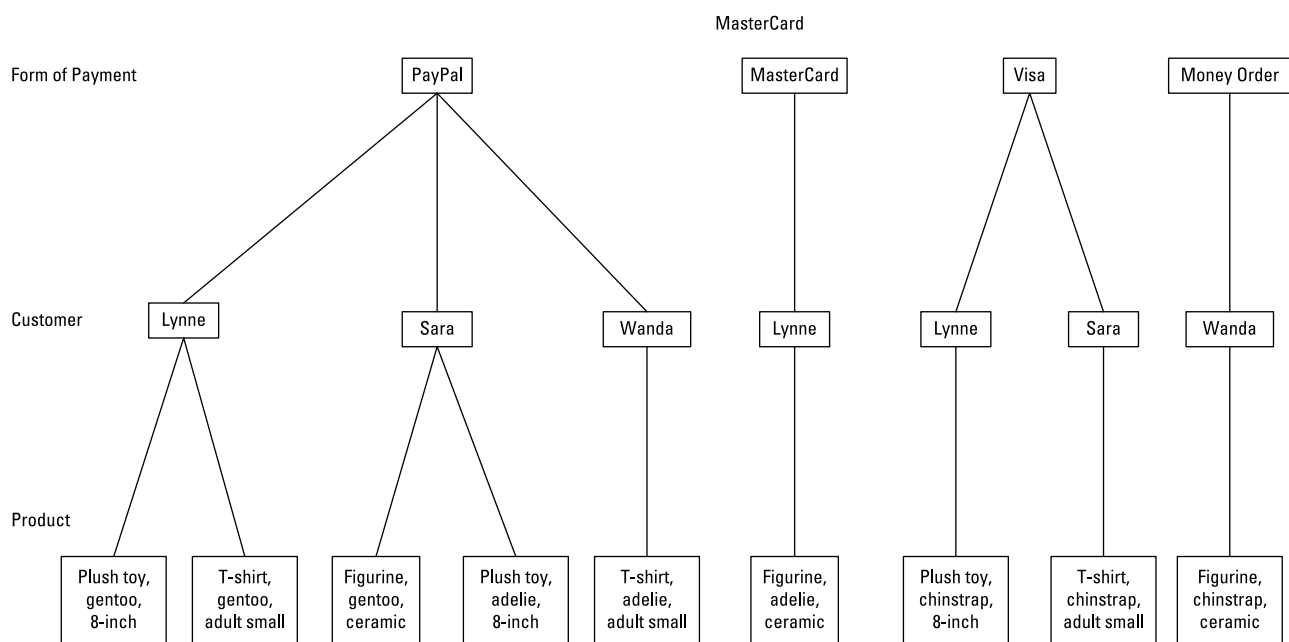


FIGURE 1-2:
A hierarchical
model of a sales
database for a
retail business.



REMEMBER

This organization is actually more complex than what is shown in Figure 1-2. Additional “trees” would hold the details about each customer and each product. This duplicate data is a waste of storage space because one copy of a customer’s data is sufficient, and so is one copy of product information.

Perhaps even more damaging than the wasted space that results from redundant data is the possibility of data corruption. Whenever multiple copies of the same data exist in a database, there is the potential for modification anomalies. A *modification anomaly* is an inconsistency in the data after a modification is made. Suppose you want to delete a customer who is no longer buying from you. If multiple copies of that customer’s data exist, you must find and delete all of them to maintain data integrity. On a slightly more positive note, suppose you just want to update a customer’s address information. If multiple copies of the customer’s data exist, you must find and modify all of them in exactly the same way to maintain data integrity. This can be a time-consuming and error-prone operation.

The network database model

The network model — the one that followed close upon the heels of the hierarchical, appearing as it did in 1969 — is almost the exact opposite of the hierarchical model. Wanting to avoid the redundancy of the hierarchical model without sacrificing too much in the way of performance, the designers of the *network model* opted for an architecture that does not duplicate items, but instead increases the number of relationships associated with some items. Figure 1-3 shows this architecture for the same data that was shown in Figure 1-2.

As you can see in Figure 1-3, the network model does not have the tree structure with one-directional flow characteristic of the hierarchical model. Looked at this way, it shows very clearly that, for example, Lynne had bought multiple products, but also that she has paid in multiple ways. There is only one instance of Lynne in this model, compared to multiple instances in the hierarchical model. However, to balance out that advantage, there are seven relationships connected to that one instance of Lynne, whereas in the hierarchical model there are no more than three relationships connected to any one instance of Lynne.



REMEMBER

The network model eliminates redundancy, but at the expense of more complicated relationships. This model can be better than the hierarchical model for some kinds of data storage tasks, but worse for others. Neither one is consistently superior to the other.

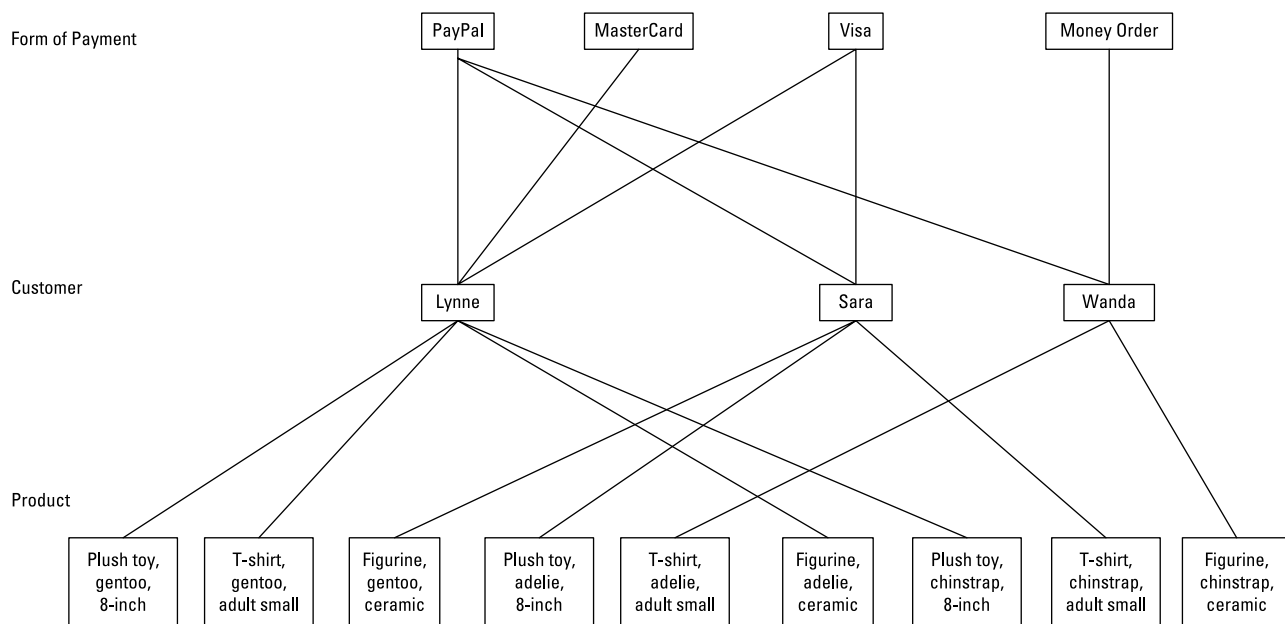


FIGURE 1-3:
A network model
of transactions at
an online store.

The relational database model

In 1970, Edgar Codd of IBM published a paper introducing the *relational database* model. Initially, database experts gave it little consideration. It clearly had an advantage over the hierarchical model in that data redundancy was minimal; it had an advantage over the network model with its relatively simple relationships. However, it had what was perceived to be a fatal flaw. Due to the complexity of the relational database engine that it required, any implementation would be much slower than a comparable implementation of either the hierarchical or the network model. As a result, it was almost ten years before the first implementation of the relational database idea hit the market.

Moore's Law had finally made relational database technology feasible. (In 1965, Gordon Moore, one of the founders of Intel, noticed that the cost of computer memory chips was dropping by half about every two years. He predicted that this trend would continue. After over 50 years, the trend is still going strong, and Moore's prediction has been enshrined as an empirical law.)

IBM delivered a relational DBMS (RDBMS) integrated into the operating system of the System 38 computer server platform in 1978, and Relational Software, Inc., delivered the first version of Oracle — the granddaddy of all standalone relational database management systems — in 1979.

Defining what makes a database relational

The original definition of a relational database specified that it must consist of two-dimensional tables of rows and columns, where the cell at the intersection of a row and column contains an atomic value (where *atomic* means not divisible into subvalues). This definition is commonly stated by saying that a relational database table may not contain any *repeating groups*. The definition also specified that each row in a table be uniquely identifiable. Another way of saying this is that every table in a relational database must have a *primary key*, which uniquely identifies a row in a database table. Figure 1-4 shows the structure of an online store database, built according to the relational model.

The relational model introduced the idea of storing database elements in two-dimensional tables. In the example shown in Figure 1-4, the Customer table contains all the information about each customer; the Product table contains all the information about each product, and the Transaction table contains all the information about the purchase of a product by a customer. The idea of separating closely related things from more distantly related things by dividing things up into tables was one of the main factors distinguishing the relational model from the hierarchical and network models.

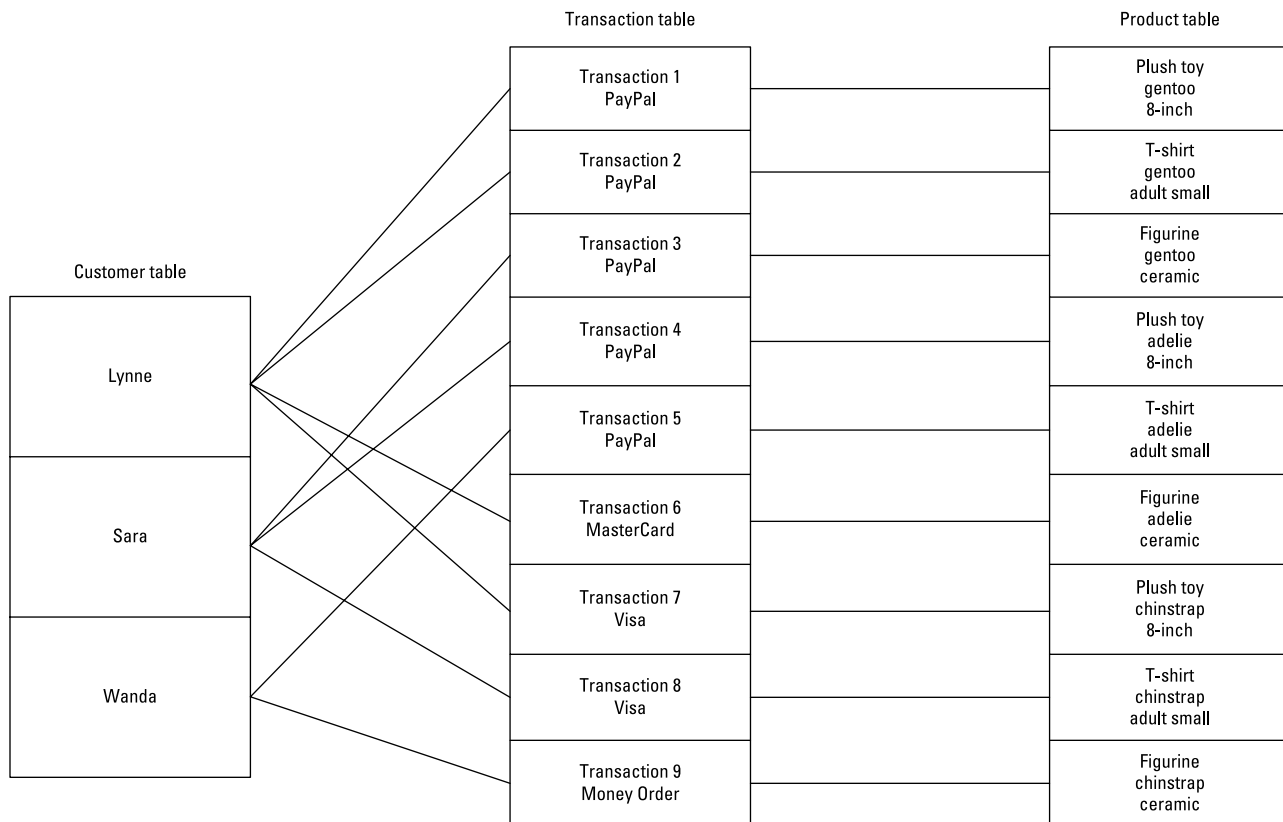


FIGURE 1-4:
A relational
model of
transactions at an
online store.

Protecting the definition of relational databases with Codd's rules

As the relational model gained in popularity, vendors of database products that were not really relational started to advertise their products as relational database management systems. To fight the dilution of his model, Codd formulated 12 rules that served as criteria for determining whether a database product was in fact relational. Codd's idea was that a database must satisfy all 12 criteria in order to be considered relational.

Codd's rules are so stringent, that even today, there is not a DBMS on the market that completely complies with all of them. However, they have provided a good goal toward which database vendors strive.

Here are Codd's 12 rules:

1. **The information rule:** Data can be represented only one way, as values in column positions within rows of a table.
2. **The guaranteed access rule:** Every value in a database must be accessible by specifying a table name, a column name, and a row. The row is specified by the value of the primary key.
3. **Systematic treatment of null values:** Missing data is distinct from specific values, such as zero or an empty string.
4. **Relational online catalog:** Authorized users must be able to access the database's structure (its *catalog*) using the same query language they use to access the database's data.
5. **The comprehensive data sublanguage rule:** The system must support at least one relational language that can be used both interactively and within application programs, that supports data definition, data manipulation, and data control functions. Today, that one language is SQL.
6. **The view updating rule:** All views that are theoretically updatable must be updatable by the system.
7. **The system must support set-at-a-time insert, update, and delete operations:** This means that the system must be able to perform insertions, updates, and deletions of multiple rows in a single operation.
8. **Physical data independence:** Changes to the way data is stored must not affect the application.
9. **Logical data independence:** Changes to the tables must not affect the application. For example, adding new columns to a table should not "break" an application that accesses the original rows.

- 10. Integrity independence:** Integrity constraints must be specified independently from the application programs and stored in the catalog. (I say a lot about integrity in Book 2, Chapter 3.)
- 11. Distribution independence:** Distribution of portions of the database to various locations should not change the way applications function.
- 12. The nonsubversion rule:** If the system provides a record-at-a-time interface, it should not be possible to use it to subvert the relational security or integrity constraints.

Over and above the original 12 rules, in 1990, Codd added one more rule:

Rule Zero: For any system that is advertised as, or is claimed to be, a relational database management system, that system must be able to manage databases entirely through its relational capabilities, no matter what additional capabilities the system may support.

Rule Zero was in response to vendors of various database products who claimed their product was a relational DBMS, when in fact it did not have full relational capability.

Highlighting the relational database model's inherent flexibility

You might wonder why it is that relational databases have conquered the planet and relegated hierarchical and network databases to niches consisting mainly of legacy customers who have been using them for more than 40 years. It's even more surprising in light of the fact that when the relational model was first introduced, most of the experts in the field considered it to be utterly uncompetitive with either the hierarchical or the network model.

One advantage of the relational model is its flexibility. The architecture of a relational database is such that it is much easier to restructure a relational database than it is to restructure either a hierarchical or network database. This is a tremendous advantage in dynamic business environments where requirements are constantly changing.

The reason database practitioners originally dissed the relational model is because the extra overhead of the relational database engine was sure to make any product based on that model so much slower than either hierarchical or network databases, as to be noncompetitive. As time has passed, Moore's Law has nullified that objection.

The object-oriented database model

Object-oriented database management systems (OODBMS) first appeared in 1980. They were developed primarily to handle nontext, nonnumeric data such as graphical objects. A relational DBMS typically doesn't do a good job with such so-called complex data types. An OODBMS uses the same data model as object-oriented programming languages such as Java, C++, and C#, and it works well with such languages.

Although object-oriented databases outperform relational databases for selected applications, they do not do as well in most mainstream applications, and have not made much of a dent in the hegemony of the relational products. As a result, I will not be saying anything more about OODBMS products.

The object-relational database model

An *object-relational database* is a relational database that allows users to create and use new data types that are not part of the standard set of data types provided by SQL. The ability of the user to add new types, called *user-defined types*, was added to the SQL:1999 specification and is available in current implementations of IBM's DB2, Oracle, and Microsoft SQL Server.

Current relational database management systems are actually object-relational database management systems rather than pure relational database management systems.

The nonrelational NoSQL model

In contrast to the relational model, a nonrelational model has been gaining adherents, particularly in the area of *cloud computing*, where databases are maintained not on the local computer or local area network, but reside somewhere on the Internet. This model, called the NoSQL model, is particularly appropriate for large systems consisting of clusters of servers, accessed over the World Wide Web. CouchDB and MongoDB are examples of DBMS products that follow this model. The NoSQL model is document based, storing all related data in the same document. Because all the related data is stored in the same place, queries for large amounts of data can be quicker than in traditional relational databases. The NoSQL model is not competitive with the SQL-based relational model for traditional reporting applications.

Why the Relational Model Won

Throughout the 1970s and into the 1980s, hierarchical- and network-based technologies were the database technologies of choice for large organizations. Oracle, the first standalone relational database system to reach the market, did not appear until 1979, and initially met with limited success.

For the following reasons, as well as just plain old inertia, relational databases caught on slowly at first:

- » **The earliest implementations of relational database management systems were slow performers.** This was due to the fact that they were required to perform more computations than other database systems to perform the same operation.
- » **Most business managers were reluctant to try something new when they were already familiar with one or the other of the older technologies.**
- » **Data and applications that already existed for an existing database system would be very difficult to convert to work with a relational DBMS.** For most organizations with an existing hierarchical or network database system, it would be too costly to make a conversion.
- » **Employees would have to learn an entirely new way of dealing with data.** This would be very costly, too.

However, things gradually started to change.

Although databases structured according to the hierarchical and network models had excellent performance, they were difficult to maintain. Structural changes to a database took a high level of expertise and a lot of time. In many organizations, backlogs of change requests grew from months to years. Department managers started putting their work on personal computers rather than going to the corporate IT department to ask for a change to a database. IT managers, fearing that their power in the organization was eroding, took the drastic step of considering relational technology.

Meanwhile, Moore's Law was inexorably changing the performance situation. In 1965, Gordon Moore of Intel noted that about every 18 months to 2 years the price of a bit in a semiconductor memory would be cut in half, and he predicted that this exponential trend would continue. A corollary of the law is that for a given cost, the performance of integrated circuit processors would double every 18 to 24

months. Both of these laws have held true for more than 50 years, although the end of the trend is in sight. In addition, the capacities and performance of hard disk storage devices have also improved at an exponential rate, paralleling the improvement in semiconductor chips.

The performance improvements in processors, memories, and hard disks combined to dramatically improve the performance of relational database systems, making them more competitive with hierarchical and network systems. When this improved performance was added to the relational architecture's inherent advantage in structural flexibility, relational database systems started to become much more attractive, even to large organizations with major investments in legacy systems. In many of these companies, although existing applications remained on their current platforms, new applications and the databases that held their data were developed using the new relational technology.

IN THIS CHAPTER

- » Picturing how to grab the data you want to grab
- » Mapping your data retrieval strategy onto a relational model
- » Using Entity-Relationship diagrams to visualize what you want
- » Understanding the relational database hierarchy

Chapter 2

Modeling a System

SQL is the language that you use to create and operate on relational databases. Before you can do that database creation, however, you must first create a conceptual model of the system to be built. In order to have any hope of developing a database system that delivers the results, performance, and reliability that the users need, you must understand, in a highly detailed way, what those needs are. Your understanding of the users' needs enables you to create a model of what they have in mind.

After perfecting the model through much dialog with the user, you need to translate the model into something that can be implemented with a relational database. This chapter takes you through the steps of taking what might be a vague and fuzzy idea in the minds of the users and transforming it into something that can be converted directly into a robust and high-performance database.

Capturing the Users' Data Model

The whole purpose of a database is to hold useful data and enable one or more people to selectively retrieve and use the data they want. Generally, before a database project is begun, interested parties have some idea of what data they want to store, and what subsets of the data they are likely to want to retrieve. More often

than not, people's ideas of what should be included in the database and what they want to get out of it are not terribly precise. Nebulous as they may be, the concepts each interested party may have in mind comes from her own data models. When all those data models from various users are combined, they become one (huge) data model.

To have any hope of building a database system that meets the needs of the users, you must understand this collective data model. In the text that follows, I give you some tips for finding and querying the people who will use the database, prioritizing requested features, and getting support from stakeholders.

Beyond understanding the data model, you must help to clarify it so that it can become the basis for a useful database system. In the “Translating the Users' Data Model to a Formal Entity-Relationship Model” section that follows this one, I tell you how to do that.

Identifying and interviewing stakeholders

The first step in discovering the users' data model is to find out who the users are. Perhaps several people will interact directly with the system. They, of course, are very interested parties. So are their supervisors, and even higher management.

But identifying the database users goes beyond the people who actually sit in front of a PC and run your database application. A number of other people usually have a stake in the development effort. If the database is going to deal with customer or vendor information, the customers and vendors are probably stakeholders, too. The IT department — the folks responsible for keeping systems up and running — is also a major stakeholder. There may be others, such as owners or major stockholders in the company. All of these people are sure to have an image in their mind of what the system ought to be. You need to find these people, interview them, and find out how they envision the system, how they expect it to be maintained, and what they want it to produce.

If the functions to be performed by the new system are already being performed, by either a manual system or an obsolete computerized system, you can ask the users to explain how their current system works. You can then ask them what they like about the current system and what they don't like. What is the motivation for moving to a new system? What desirable features are missing from what they have now? What annoying aspects of the current system are frustrating them? Try to gain as complete an understanding of the current situation as possible.

Reconciling conflicting requirements

Just as the set of stakeholders will be diverse, so will their ideas of what the system should be and do. If such ideas are not reconciled, you are sure to have a disaster on your hands. You run the risk of developing a system that is not satisfactory to anybody.

It is your responsibility as the database developer to develop a consensus. You are the only independent, outside party who does not have a personal stake in what the system is and does. As part of your responsibility, you'll need to separate the stated requirements of the stakeholders into three categories, as follows:

- » **Mandatory:** A feature that is absolutely essential falls into this category. The system would be of limited value without it.
- » **Significant:** A feature that is important and that adds greatly to the value of the system belongs in this category.
- » **Optional:** A feature that would be nice to have, but is not actually needed, falls into this category.

Once you have appropriately categorized the want lists of the stakeholders, you are in a position to determine what is really required, and what is possible within the allotted budget and development time. Now comes the fun part. You must convince all the stakeholders that their cherished features that fall into the third category (optional), must be deleted or changed if they conflict with someone else's first-category or second-category feature. Of course, politics also intrudes here. Some stakeholders have more clout than others. You must be sensitive to this. Sometimes the politically acceptable solution is not exactly the same as the technically optimal solution.

Obtaining stakeholder buy-in

One way or another, you will have to convince all the stakeholders to agree on one set of features that will be included in the system you are planning to build. This is critical. If the system does not adequately meet the needs of all those for whom it is being built, it is not a success. You must get the agreement of everyone that the system you propose meets their needs. Get it in writing. Enumerate everything that will be provided in a formal Statement of Requirements, and then have every stakeholder sign off on it. This will potentially save you from much grief later on.

DATABASE DEVELOPERS ARE LIKE ARMY DOCTORS

Battleground field hospitals make use of a technique called *triage* to allocate their limited resources in the most beneficial way. When people are brought in for treatment, they are examined to determine the extent of their injuries. After the examination, each is placed into one of three categories:

- The person has critical wounds and must receive treatment immediately or he will die.
- The person has serious wounds, but they are not immediately life-threatening. The doctors can afford to let this person wait while patients with more serious injuries are treated.
- The person is so badly wounded that no treatment available will save her.

Patients in the first category are treated immediately. Patients in the second category are treated as soon as circumstances permit. Patients in the third category are made as comfortable as possible, but treated only for pain.

Translating the Users' Data Model to a Formal Entity-Relationship Model

After you outline a coherent users' data model in a clear, concise, concrete form, the real work begins. Somehow, you must transform that model into a relational model that serves as the basis for a database. In most cases, a users' data model is not in a form that can be directly translated into a relational model. A helpful technique is to first translate it into one of several formal modeling systems that clarify the various entities in the users' model and the relationships between them. Probably the most popular of those formal modeling techniques is the Entity-Relationship (ER) model. Although there are other formal modeling systems, I focus on the ER model because it is the most widespread and thus easily understood by most database professionals.

Graphing tools — Microsoft Visio, for example — make provision for drawing representations of an ER model. I guess I am old fashioned in that I prefer to draw them by hand on paper with a pencil. This gives me a little more flexibility in how I arrange the elements and how I represent them.

SQL is the international standard language for *communicating* with relational databases. Before you can fully appreciate SQL, you must understand the *structure* of well-designed relational databases. In order to design a relational database properly — in hopes that it will be reliable as well as giving the level of performance you need — you must have a good understanding of database structure. This is best achieved through database modeling, and the most widely used model is the Entity-Relationship model.

Entity-Relationship modeling techniques

In 1976, six years after Dr. Codd published the relational model, Dr. Peter Chen published a paper in the reputable journal *ACM Transactions on Database Systems*, introducing the Entity-Relationship (ER) model, which represented a conceptual breakthrough because it provided a means to translate a users' data model into a relational model.

Back in 1976, the relational model was still nothing more than a theoretical construct. It would be three more years before the first standalone relational database product (Oracle) appeared on the market.



REMEMBER

The ER model was an important factor in turning theory into practice because one of the strengths of the ER model is its generality. ER models can represent a wide variety of different systems. For example, an ER model can represent a physical system as big and complex as a fleet of cruise ships, or as small as the collection of livestock maintained by a gentleman farmer on his two acres of land.

Any Entity-Relationship model, big or small, consists of four major components: entities, attributes, identifiers, and relationships. I examine each one of these concepts in turn.

Entities

Dictionaries tell you that an *entity* is something that has a distinct, separate existence. It could be a material entity, such as the Great Pyramid of Giza, or an abstract entity, such as a tetrahedron. Just about any distinct, separate thing that you can think of qualifies as being an entity. When used in a database context, an *entity* is something that the user can identify and that she wants to keep track of.

A group of entities with common characteristics is called an *entity class*. Any one example of an entity class is an *entity instance*. A common example of an entity class for most organizations is the EMPLOYEE entity class. An example of an *instance* of that entity class is a particular employee, such as Duke Kahanamoku.

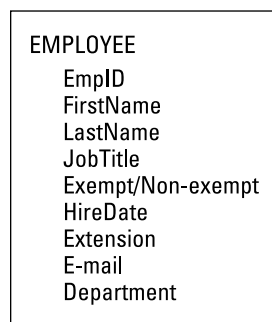
In the previous paragraph, I spell out EMPLOYEE with all caps. This is a convention that I will follow throughout this book so that you can readily identify entities in the ER model. I follow the same convention when I refer to the tables in the relational model that correspond to the entities in the ER model. Other sources of information on relational databases that you read may use all lowercase for entities, or an initial capital letter followed by lowercase letters. There is no standard. The database management systems that will be processing the SQL that is based on your models do not care about capitalization. Agreeing to a standard is meant to reduce confusion among the people dealing with the models and with the code generated based on those models — the models themselves don't care.

Attributes

Entities are things that users can identify and want to keep track of. However, the users probably don't want to use up valuable storage space keeping track of every conceivable aspect of an entity. Some aspects are of more interest than others. For example, in the EMPLOYEE model, you probably want to keep track of such things as first name, last name, and job title. You probably do not want to keep track of the employee's favorite surfboard manufacturer or favorite musical group.

In database-speak, aspects of an entity are referred to as *attributes*. Figure 2-1 shows an example of an entity class — including the kinds of attributes you'd expect someone to highlight for this particular (EMPLOYEE) entity class. Figure 2-2 shows an example of an instance of the EMPLOYEE entity class. EmpID, FirstName, LastName, and so on are attributes.

FIGURE 2-1:
EMPLOYEE, an
example of an
entity class.



Identifiers

In order to do anything meaningful with data, you must be able to distinguish one piece of data from another. That means each piece of data must have an identifying characteristic that is unique. In the context of a relational database, a “piece of data” is a row in a two-dimensional table. For example, if you were to construct

an EMPLOYEE table using the handy EMPLOYEE entity class and attributes spelled out back in Figure 2-1, the row in the table describing Duke Kahanamoku would be the piece of data, and the EmpID attribute would be the identifier for that row. No other employee will have the same EmpID as the one that Duke has.

FIGURE 2-2:
Duke Kahanamoku, an example of an instance of the EMPLOYEE entity class.

EMPLOYEE
172850
Duke
Kahanamoku
Cultural ambassador
E
01/01/2002
10
duck@surfboardsrus.com
Public Relations

In this example, EmpID is not just an identifier — it is a unique identifier. There is one and only one EmpID that corresponds to Duke Kahanamoku. Nonunique identifiers are also possible. For example, a FirstName of Duke does not uniquely identify Duke Kahanamoku. There might be another employee named Duke — Duke Snyder, let’s say. Having an attribute such as EmpID is a good way to guarantee that you are getting the specific employee you want when you search the database.

Another way, however, is to use a *composite identifier*, which is a combination of several attributes that together are sufficient to uniquely identify a record. For example, the combination of FirstName and LastName would be sufficient to distinguish Duke Kahanamoku from Duke Snyder, but would not be enough to distinguish him from his father, who, let’s say, has the same name and is employed at the same company. In such a case, a composite identifier consisting of FirstName, LastName, and BirthDate would probably suffice.

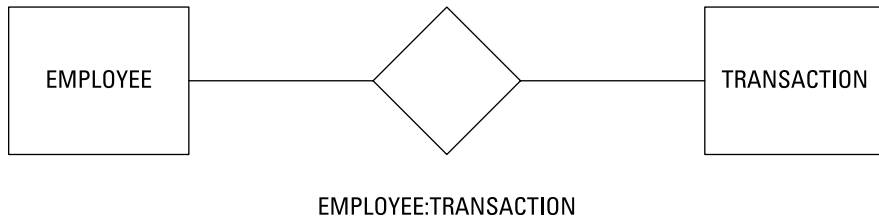
Relationships

Any nontrivial relational database contains more than one table. When you have more than one table, the question arises as to how the tables relate to each other. A company might have an EMPLOYEE table, a CUSTOMER table, and a PRODUCT table. These become related when an employee sells a product to a customer. Such a sales transaction can be recorded in a TRANSACTION table. Thus the EMPLOYEE, CUSTOMER, and PRODUCT tables are related to each other via the TRANSACTION table. Relationships such as these are key to the way relational databases operate. Relationships can differ in the number of entities that they relate.

DEGREE-TWO RELATIONSHIPS

Degree-two relationships are ones that relate one entity directly to one other entity. `EMPLOYEE` is related to `TRANSACTION` by a degree-two relationship, also called a *binary relationship*. `CUSTOMER` is also related to `TRANSACTION` by a binary relationship, as is `PRODUCT`. Figure 2-3 shows a diagram of a degree-two relationship.

FIGURE 2-3:
An `EMPLOYEE`:
`TRANSACTION`
relationship.



Degree-two relationships are the simplest possible relationships, and happily, just about any system that you are likely to want to model consists of entities connected by degree-two relationships, although more complex relationships are possible.

There are three kinds of binary (degree-two) relationships:

- » **One-to-one (1:1) relationship:** Relates one instance of one entity class (a group of entities with common characteristics) to one instance of a second entity class.
- » **One-to-many (1:N) relationship:** Relates one instance of one entity class to multiple instances of a second entity class.
- » **Many-to-many (N:M) relationship:** Relates multiple instances of one entity class to multiple instances of a second entity class.

Figure 2-4 is a diagram of a one-to-one relationship between a person and that person's driver's license. A person can have one and only one driver's license, and a driver's license can apply to one and only one person. This database would contain a `PERSON` table and a `LICENSE` table (both are entity classes), and the Duke Snyder instance of the `PERSON` table has a one-to-one relationship with the OR31415927 instance of the `LICENSE` table.

Figure 2-5 is a diagram of a one-to-many relationship between the `PERSON` entity class and the traffic violation `TICKET` entity class. A person can be served with multiple tickets, but a ticket can apply to one and only one person.

FIGURE 2-4:
A one-to-one
relationship
between PERSON
and LICENSE.

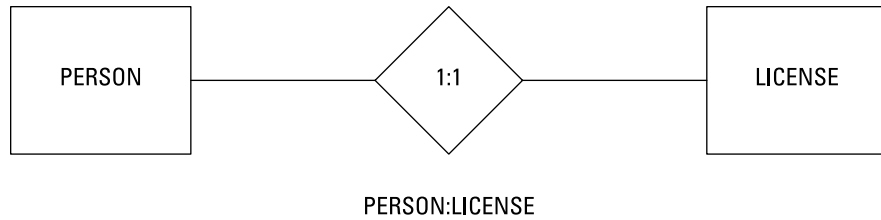
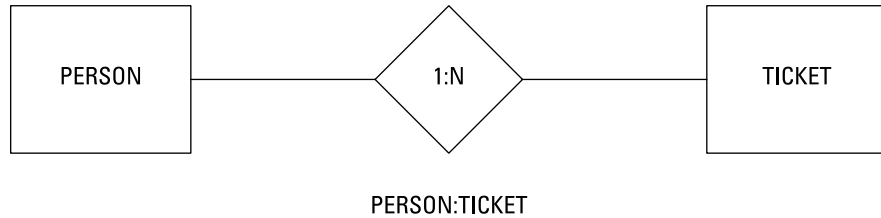


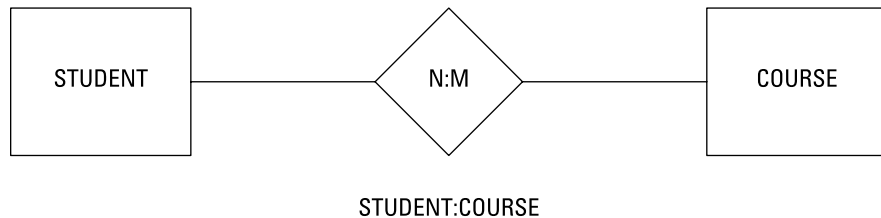
FIGURE 2-5:
A one-to-many
relationship
between PERSON
and TICKET.



When this part of the ER model is translated into database tables, there will be a row in the PERSON table for each person in the database. There could be zero, one, or multiple rows in the TICKET table corresponding to each person in the PERSON table.

Figure 2-6 is a diagram of a many-to-many relationship between the STUDENT entity class and the COURSE entity class, which holds the route a person takes on her drive to work. A person can take one of several routes from home to work, and each one of those routes can be taken by multiple people.

FIGURE 2-6:
A many-to-many
relationship
between
STUDENT and
COURSE.



Many-to-many relationships can be very confusing and are not well represented by the two-dimensional table architecture of a relational database. Consequently, such relationships are almost always converted to simpler one-to-many relationships before they are used to build a database.

COMPLEX RELATIONSHIPS

Degree-three relationships are possible, but rarely occur in practice. Relationships of degree higher than three probably mean that you need to redesign your system

to use simpler relationships. An example of a degree-three relationship is the relationship between a musical composer, a lyricist, and a song. Figure 2-7 shows a diagram of this relationship.

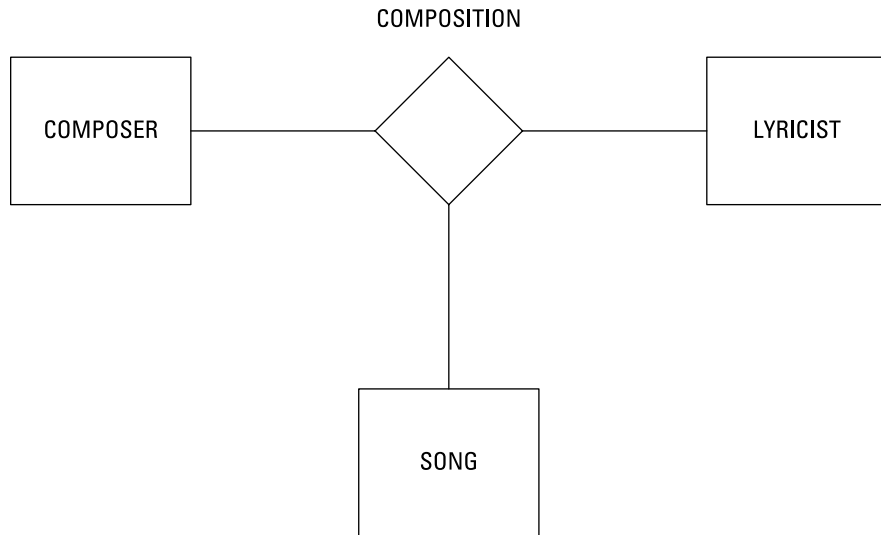


FIGURE 2-7:
The COMPOSER:
SONG: LYRICIST
relationship.



TIP

Although it is possible to build a system with such relationships, it is probably better in most cases to restructure the system in terms of binary relationships.

Drawing Entity-Relationship diagrams

I've always found it easier to understand relationships between things if I see a diagram instead of merely looking at sentences describing the relationships. Apparently a lot of other people feel the same way; systems represented by the Entity-Relationship model are universally depicted in the form of diagrams. A few simple examples of such *ER diagrams*, as I refer to them, appear in the previous section. In this section, I introduce some concepts that add detail to the diagrams.

One of those concepts is *cardinality*. In mathematics, cardinality is the number of elements in a set. In the context of relational databases, a relationship between two tables has two cardinalities of interest: the cardinality — number of elements — associated with the first table and the cardinality — you guessed it, the number of elements — associated with the second table. We look at these cardinalities two primary ways: maximum cardinality and minimum cardinality, which I tell you about in the following sections. (Cardinality only becomes truly important when you are dealing with queries that pull data from multiple tables. I discuss such queries in Book 3, Chapters 3 and 4.)

Maximum cardinality

The *maximum cardinality* of one side of a relationship shows the largest number of entity instances that can be on that side of the relationship.

For example, the ER diagram's representation of maximum cardinality is shown back in Figures 2-4, 2-5, and 2-6. The diamond between the two entities in the relationship holds the two maximum cardinality values. Figure 2-4 shows a one-to-one relationship. In the example, a person is related to that person's driver's license. One driver can have at most one license, and one license can belong at most to one driver. The maximum cardinality on both sides of the relationship is one.

Figure 2-5 illustrates a one-to-many relationship. When relating a person to the tickets he has accumulated, each ticket belongs to one and only one driver, but a driver may have more than one ticket. The number of tickets above one is indeterminate, so it is represented by the variable N.

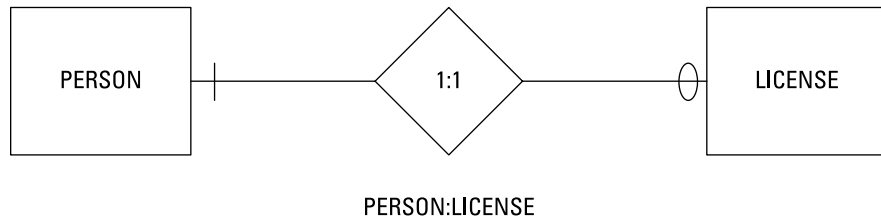
Figure 2-6 shows a many-to-many relationship. The maximum cardinality on the STUDENT side is represented by the variable N, and the maximum cardinality on the COURSE side is represented by the variable M because although both the number of students and the number of courses are more than one, they are not necessarily the same. You might have 350 different students that take any of 45 courses, for example.

Minimum cardinality

Whereas the maximum cardinality of one side of a relationship shows the largest number of entity instances that can be on that side of the relationship, the *minimum cardinality* shows the least number of entity instances that can be on that side of the relationship. In some cases, the least number of entity instances that can be on one side of a relationship can be zero. In other cases, the minimum cardinality could be one or more.

Refer to the relationship in Figure 2-4 between a person and that person's driver's license. The minimum cardinalities in the relationship depend heavily on subtle details of the users' data model. Take the case where a person has been a licensed driver, but due to excessive citations, his driver's license has been revoked. The person still exists, but the license does not. If the users' data model stipulates that the person is retained in the PERSON table, but the corresponding row is removed from the LICENSE table, the minimum cardinality on the PERSON side is one, and the minimum cardinality on the LICENSE side is zero. Figure 2-8 shows how minimum cardinality is represented in this example.

FIGURE 2-8:
ER diagram
showing
minimum
cardinality, where
a person must
exist, but his
corresponding
license need
not exist.

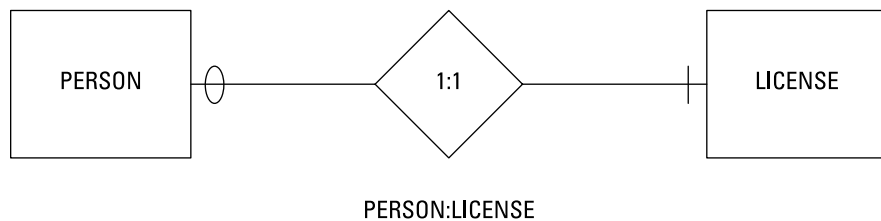


The slash mark on the PERSON side of the diagram denotes a minimum cardinality of *mandatory*, meaning at least one instance must exist. The oval on the LICENSE side denotes a minimum cardinality of *optional*, meaning at least one instance need not exist.

For this one-to-one relationship, a given person can correspond to at most one license, but may correspond to none. A given license *must* correspond to one person.

If only life were that simple. . . . Remember that I said that minimum cardinality depends subtly on the users' data model? What if the users' data model were slightly different, based on another possible case? Suppose a person has a very good driving record and a valid driver's license in her home state of Washington. Next, suppose that she accepts a position as a wildlife researcher on a small island that has no roads and no cars. She is no longer a driver, but her license will remain valid until it expires in a few years. This is the reverse case of what is shown in Figure 2-8; a license exists, but the corresponding driver does not (at least as far as the state of Washington is concerned). Figure 2-9 shows this situation.

FIGURE 2-9:
ER diagram
showing
minimum
cardinality, where
a license must
exist, but its
corresponding
person need
not exist.



The lesson to take home from this example is that minimum cardinality is often difficult to determine. You'll need to question the users very carefully and explore unusual cases such as those cited previously before deciding how to model minimum cardinality.

If the minimum cardinality of one side of a relationship is mandatory, that means the cardinality of that side is at least one, but might be more. Suppose, for example, you were modeling the relationship between a basketball team in a city league and its players. A person cannot be a basketball player in the league and thus in the database unless she is a member of a basketball team in the league, so the minimum cardinality on the TEAM side is mandatory, and in fact is one. This assumes that the users' data model states that a player cannot be a member of more than one team. Similarly, it is not possible for a basketball team to exist in the database unless it has at least five players. This means that the minimum cardinality on the PLAYER side is also mandatory, but in this case is five. Once again, depending on the users' data model, the rule might be that a team cannot exist in the database unless it has at least five players. The minimum cardinality of the PLAYER side of the relationship is five.



TIP

Primarily, you are interested in whether the minimum cardinality on a side of a relationship is either mandatory or optional and less interested in whether a mandatory minimum cardinality has a value of one or more than one. The difference between mandatory and optional is the difference between whether an entity exists or not. The difference between existence and nonexistence is substantial. In contrast, the difference between one and five is just a matter of degree. Both cases refer to a mandatory minimum cardinality. For most applications, the difference between one mandatory value and another does not matter.

Understanding advanced ER model concepts

In the previous sections of this chapter, I talk about entities, relationships, and cardinality. I point out that subtle differences in the way users model their system can modify the way minimum cardinality is modeled. These concepts are a good start, and are sufficient for many simple systems. However, more complex situations are bound to arise. These call for extensions of various sorts to the ER model. To limber up your brain cells so you can tackle such complexities, take a look at a few of these situations and the extensions to the ER model that have been created to deal with them.

Strong entities and weak entities

All entities are not created equal. Some are stronger than others. An entity that does not depend on any other entity for its existence is considered a *strong entity*. Consider the sample ER model in Figure 2-10. All the entities in this model are strong, and I tell you why in the paragraphs that follow.

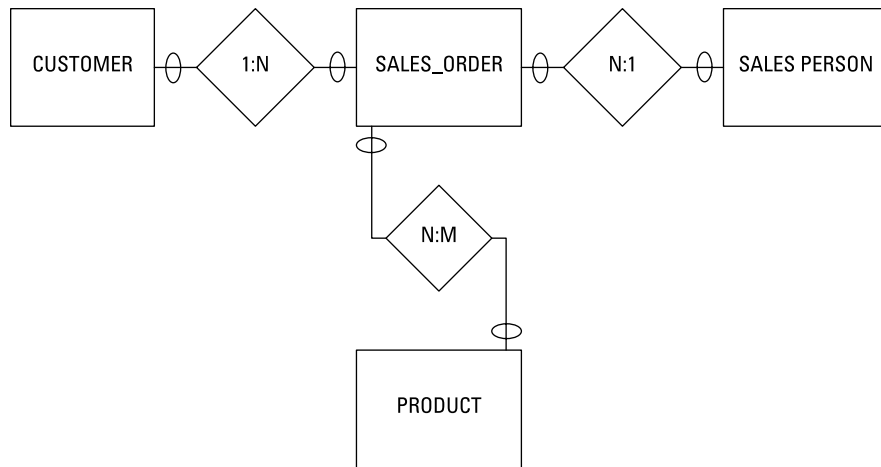


FIGURE 2-10:
The ER model
for a retail
transaction
database.

To get this “depends on” business straight, do a bit of a thought experiment. First, consider maximum cardinality. A customer (whose data lies in the CUSTOMER table) can make multiple purchases, each one recorded on a sales order (the details of which show up in the SALES_ORDER table). A SALESPERSON can make multiple sales, each one recorded on a SALES_ORDER. A SALES_ORDER can include multiple PRODUCTS, and a PRODUCT can appear on multiple SALES_ORDERS.

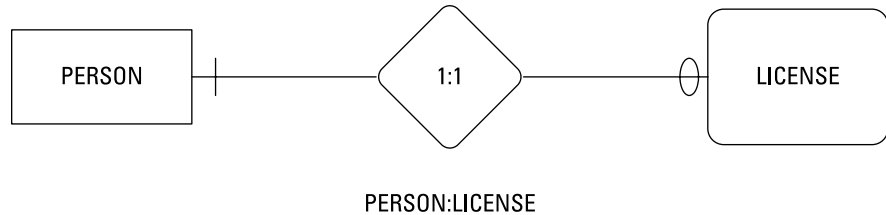
Minimum cardinality may be modeled a variety of ways, depending on how the users’ data model views things. For example, a person might be considered a customer (someone whose data appears in the CUSTOMER table) even before she buys anything because the store received her information in a promotional campaign. An employee might be considered a salesperson as soon as he is hired, even though he hasn’t sold anything yet. A sales order might exist before it lists any products, and a product might exist on the shelves before any of them have been sold. According to this model, all the minimum cardinalities are optional. A different users’ data model could mandate that some of these relationships be mandatory.

In a model such as the one described, where all the minimum cardinalities are optional, none of the entities depends on any of the other entities for its existence. A customer can exist without any associated sales orders. An employee can exist without any associated sales orders. A product can exist without any associated sales orders. A sales order can exist in the order pad without any associated customer, salesperson, or product. In this arrangement, all these entities are classified as *strong entities*. They all have an independent existence. Strong entities are represented in ER diagrams as rectangles with sharp corners.

Not all entities are strong, however. Consider the case shown in Figure 2-11. In this model, a driver’s license cannot exist unless the corresponding driver exists.

The license is *existence-dependent* upon the driver. Any entity that is existence-dependent on another entity is a *weak entity*. In an ER diagram, a weak entity is represented with a box that has rounded corners. The diamond that shows the relationship between a weak entity and its corresponding strong entity also has rounded corners. Figure 2-11 shows this representation.

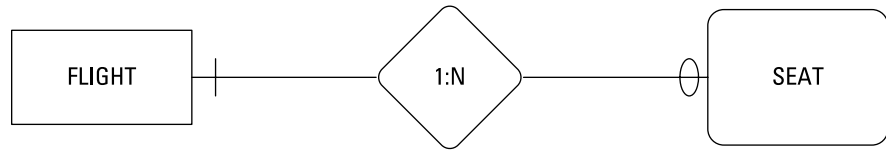
FIGURE 2-11:
A PERSON:
LICENSE
relationship,
showing LICENSE
as a weak entity.



ID-dependent entities

A weak entity cannot exist without a relationship to a strong entity. A special case of a weak entity is one that depends on a strong entity not only for its existence, but also for its identity — this is called an *ID-dependent entity*. One example of an ID-dependent entity is a seat on an airliner flight. Figure 2-12 illustrates the relationship.

FIGURE 2-12:
The SEAT is
ID-dependent
on FLIGHT via
the FLIGHT: SEAT
relationship.



A seat number, for example 23-A, does not completely identify an airline seat. However, seat 23-A on Hawaiian Airlines flight 25 from PDX to HNL, on May 2, 2019, *does* completely identify a particular seat that a person can reserve. Those additional pieces of information are all attributes of the FLIGHT entity — the strong entity without whose existence the weak SEAT entity would basically be just a gleam in someone's eye.

Supertype and subtype entities

In some databases, you may find some entity classes that might actually share attributes with other entity classes, instead of being as dissimilar as customers and products. One example might be an academic community. There are a number of people in such a community: students, faculty members, and nonacademic staff. All those people share some attributes, such as name, home address, home

telephone number, and email address. However, there are also attributes that are not shared. A student would also have attributes of grade point average, class standing, and advisor. A faculty member would have attributes of department, academic rank, and phone extension. A staff person would have attributes of job category, job title, and phone extension.

You can create an ER model of this academic community by making STUDENT, FACULTY, and STAFF all *subtypes* of the *supertype* COMMUNITY. Figure 2-13 shows the relationships.

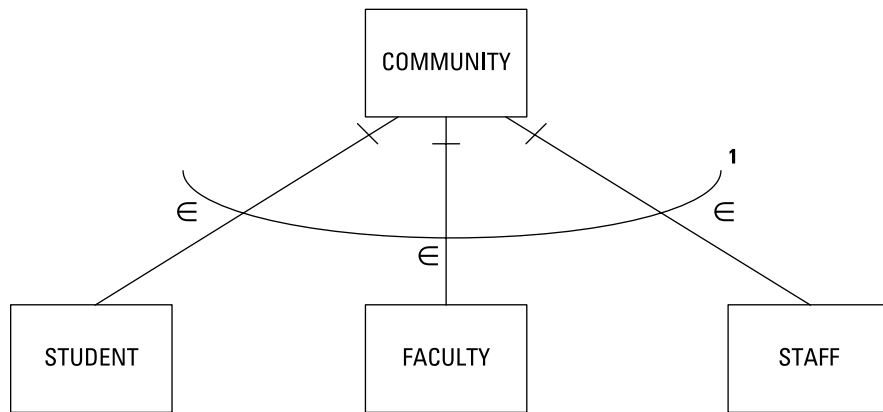


FIGURE 2-13: The COMMUNITY supertype entity with STUDENT, FACULTY, and STAFF subtype entities.

COMMUNITY contains:
Name
Home Address
Telephone Number
E-mail Address

STUDENT contains:
Grade Point Average
Advisor
Class Standing

FACULTY contains:
Academic Rank
Department
Phone Extension

STAFF contains:
Job Category
Job Title
Phone Extension

Supertype/subtype relationships borrow the concept of *inheritance* from object-oriented programming. The attributes of the *supertype entity* are inherited by the subtype entities. Each *subtype entity* has additional attributes that it does not necessarily share with the other subtype entities. In the example, everyone in the community has a name, a home address, a telephone number, and an email address. However, only students have a grade point average, an advisor, and a class standing. Similarly, only a faculty member can have an academic rank, and only a staff member can have a job title.

Some aspects of Figure 2-13 require a little additional explanation. The € next to each relationship line signifies that the lower entity is a subtype of the higher entity, so STUDENT, FACULTY, and STAFF are subtypes of COMMUNITY. The curved arc with a number 1 at the right end represents the fact that every member

of the `COMMUNITY` must be a member of one of the subtype entities. In other words, you cannot be a member of the community unless you are either a student, or a faculty member, or a staff member. It is possible in some models that an element could be a member of a supertype without being a member of any of the subtypes. However, that is not the case for this example.

The supertype and subtype entities in the ER model correspond to supertables and subtables in a relational database. A supertable can have multiple subtables and a subtable can also have multiple supertables. The relationship between a supertable and a subtable is always one-to-one. The supertable/subtable relationship is created with an `SQL CREATE` command. I give an example of an ER model that incorporates a supertype/subtype structure later in this chapter.

Incorporating business rules

Business rules are formal statements about how an organization does business. They typically differ from one organization to another. For example, one university may have a rule that a faculty member must hold a PhD degree. Another university could well have no such rule.

Sometimes you may not find important business rules written down anywhere. They may just be things that everyone in the organization understands. It is important to conduct an in-depth interview of everyone involved to fish out any business rules that people failed to mention when the job of creating the database was first described to you.

A simple example of an ER model

In this section, as an example, I apply the principles of ER models to a hypothetical web-based business named Gentoo Joyce that sells apparel items with penguin motifs, such as T-shirts, scarves, and dresses. The business displays its products and takes credit card orders on its website. There is no brick and mortar store. Fulfillment is outsourced to a fulfillment house, which receives and warehouses products from vendors, and then, upon receiving orders from Gentoo Joyce, ships the orders to customers.

The website front end consists of pages that include descriptions and pictures of the products, a shopping cart, and a form for capturing customer and payment information. The website back end holds a database that stores customer, transaction, inventory, and order shipment status information. Figure 2-14 shows an ER diagram of the Gentoo Joyce system. It is an example typical of a boutique business.

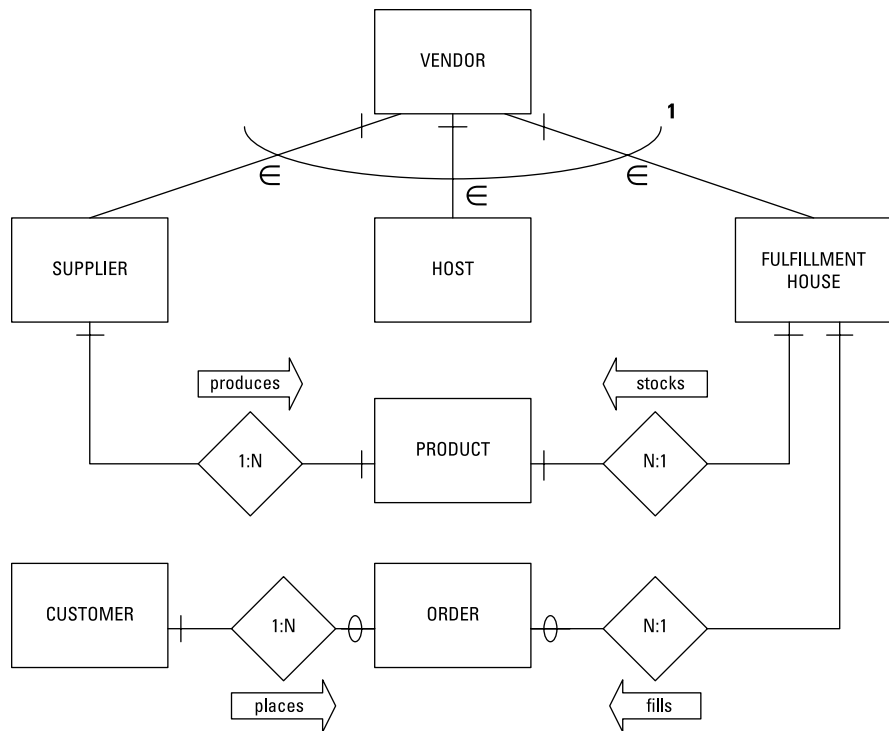


FIGURE 2-14:
An ER diagram
of a small,
web-based
retail business.

Gentoo Joyce buys goods and services from three kinds of vendors: product suppliers, web hosting services, and fulfillment houses. In the model, **VENDOR** is a supertype of **SUPPLIER**, **HOST**, and **FULFILLMENT_HOUSE**. Some attributes are shared among all the vendors; these are assigned to the **VENDOR** entity. Other attributes are not shared and are instead attributes of the subtype entities.



REMEMBER

This is only one of several possible models for the Gentoo Joyce business. Another possibility would be to include all providers in a **VENDOR** entity with more attributes. A third possibility would be to have no **VENDOR** entity, but separate **SUPPLIER** and **FULFILLMENT_HOUSE** entities, and to just consider a host as a supplier.

A many-to-many relationship exists between **SUPPLIER** and **PRODUCT** because a supplier may provide more than one product, and a given product may be supplied by more than one supplier. Similarly, any given product will (hopefully) appear on multiple orders, and an order may include multiple products. Such many-to-many relationships can be problematic. I discuss how to handle such problems in Book 2.

The other relationships in the model are one-to-many. A customer can place many orders, but each order comes from one and only one customer. A fulfillment

house can stock multiple products, but each product is stocked by one and only one fulfillment house.

A slightly more complex example

The Gentoo Joyce system that I describe in the preceding section is an easy-to-understand example, similar to what you often find in database textbooks. Most real-world systems are much more complex. I don't try to show a genuine, real-world system here, but to move at least one step in that direction, I model the fictitious Clear Creek Medical Clinic (CCMC). As I discuss in Book 2 as well as earlier in this chapter, one of the first things to do when assigned the project of creating a database for a client is to interview everyone who has a stake in the system, including management, users, and anyone else who has a say in how things are run. Listen carefully to these people and discern how they model in their minds the system they envision. Find out what information they need to capture and what they intend to do with it.

CCMC employs doctors, nurses, medical technologists, medical assistants, and office workers. The company provides medical, dental, and vision benefits to employees and their dependents. The doctors, nurses, and medical technologists must all be licensed by a recognized licensing authority. Medical assistants may be certified, but need not be. Neither licensure nor certification is required of office workers.

Typically, a patient will see a doctor, who will examine the patient, and then order one or more tests. A medical assistant or nurse may take samples of the patient's blood, urine, or both, and take the samples to the laboratory. In the lab, a medical technologist performs the tests that the doctor has ordered. The results of the tests are sent to the doctor who ordered them, as well as to perhaps one or more consulting physicians. Based on the test results, the primary doctor, with input from the consulting physicians, makes a diagnosis of the patient's condition and prescribes a treatment. A nurse then administers the prescribed treatment.

Based on the descriptions of the envisioned system, as described by the interested parties (called stakeholders), you can come up with a proposed list of entities. A good first shot at this is to list all the nouns that were used by the people you interviewed. Many of these will turn out to be entities in your model, although you may end up classifying some of those nouns as attributes of entities. For this example, say you generated the following list:

- Employee
- Office worker
- Doctor (physician)

Nurse
Medical technologist
Medical assistant
Benefits
Dependents
Patients
Doctor's license
Nurse's license
Medical technologist's license
Medical assistant's certificate
Examination
Test order
Test
Test result
Consultation
Diagnosis
Prescription
Treatment

In the course of your interviews of the stakeholders, you found that one of the categories of things to track is employees, but there are several different employee classifications. You also found that there are benefits, and those benefits apply to dependents as well as to employees. From this, you conclude that `EMPLOYEE` is an entity and it is a supertype of the `OFFICE_WORKER`, `DOCTOR`, `NURSE`, `MEDTECH`, and `MEDASSIST` entities. A `DEPENDENT` entity also should fit into the picture somewhere.

Although doctors, nurses, and medical technologists all must have current valid licenses, because a license applies to one and only one professional and each professional has one and only one license, it makes sense for those licenses to be attributes of their respective `DOCTOR`, `NURSE`, and `MEDTECH` entities rather than to be entities in their own right. Consequently, there is no `LICENSE` entity in the CCMC ER model.

PATIENT clearly should be an entity, as should EXAMINATION, TEST, TESTORDER, and RESULT. CONSULTATION, DIAGNOSIS, PRESCRIPTION, and TREATMENT also deserve to stand on their own as entities.

After you have decided what the entities are, you can start thinking about how they relate to each other. You may be able to model each relationship in one of several ways. This is where the interviews with the stakeholders are critical. The model you arrive at must be consistent with the organization's business rules, both those written down somewhere and those that are understood by everyone, but not usually talked about. Figure 2-15 shows one possible way to model this system.

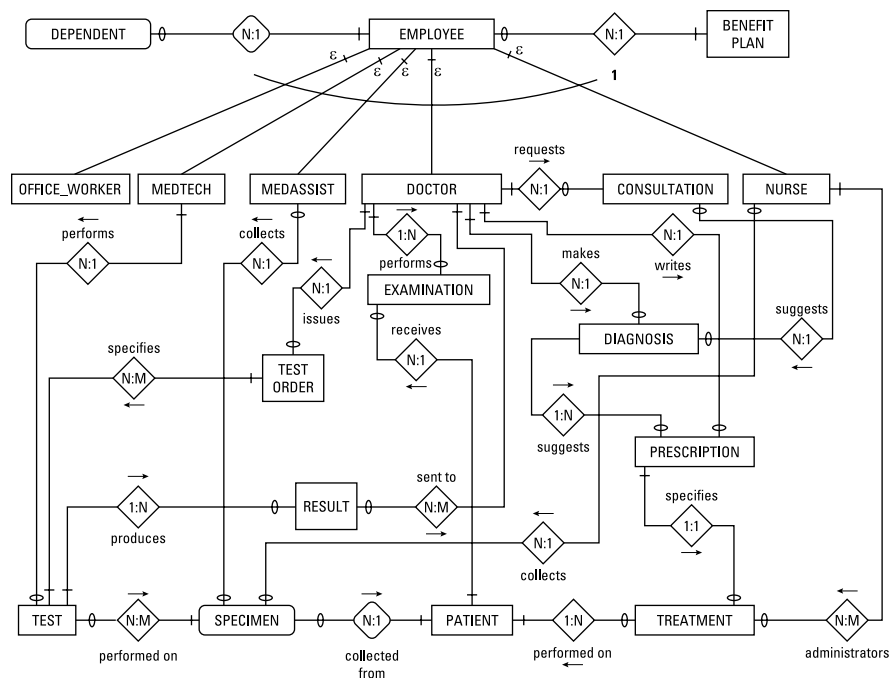


FIGURE 2-15:
The ER diagram
for Clear Creek
Medical Clinic.

From this diagram, you can extract certain facts:

- » An employee can have zero, one, or multiple dependents, but each dependent is associated with one and only one employee. (Business rule: If both members of a married couple work for the clinic, for insurance purposes, the dependents are associated with only one of them.)
- » An employee must be either an office worker, a doctor, a nurse, a medical technologist, or a medical assistant. (Business rule: An office worker cannot,

for example, also be classified as a medical assistant. Only one job classification is permitted.)

- » A doctor can perform many examinations, but each examination is performed by one and only one doctor. (Business rule: If more than one doctor is present at a patient examination, only one of them takes responsibility for the examination.)
- » A doctor can issue many test orders, but each test order can specify one and only one test.
- » A medical assistant or a nurse can collect multiple specimens from a patient, but each specimen is from one and only one patient.
- » A medical technologist can perform multiple tests on a specimen, and each test can be applied to multiple specimens.
- » A test may have one of several results; for example, positive, negative, below normal, normal, above normal, as well as specific numeric values. However, each such result applies to one and only one test.
- » A test result can be sent to one or more doctors. A doctor can receive many test results.
- » A doctor may request a consultation with one or more other doctors.
- » A doctor may make a diagnosis of a patient's condition, based on test results and possibly on one or more consultations.
- » A diagnosis could suggest one or more prescriptions.
- » A doctor can write many prescriptions, but each prescription is written by one and only one doctor for one and only one patient.
- » A doctor may order a treatment, to be administered to a patient by a nurse.

Often after drawing an ER diagram, and then determining all the things that the diagram implies by compiling a list such as that given here, the designer finds missing entities or relationships, or realizes that the model does not accurately represent the way things are actually done in the organization. Creating the model is an iterative process of progressively modifying the diagram until it reflects the desired system as closely as possible. (*Iterative* here meaning doing it over and over again until you get it right — or as right as it will ever be.)

Problems with complex relationships

The Clear Creek Medical Clinic example in the preceding section contains some many-to-many relationships, such as the relationship between TEST and

SPECIMEN. Multiple tests can be run on a single specimen, and multiple specimens, taken from multiple patients, can all be run through the same test.

That all sounds quite reasonable, but in point of fact there's a bit of a problem when it comes to storing the relevant information. If the TEST entity is translated into a table in a relational database, how many columns should be set aside for specimens? Because you don't know how many specimens a test will include, and because the number of specimens could be quite large, it doesn't make sense to allocate space in the TEST table to show that the test was performed on a particular specimen.

Similarly, if the SPECIMEN entity is translated into a table in a relational database, how many columns should you set aside to record the tests that might be performed on it? It doesn't make sense to allocate space in the SPECIMEN table to hold all the tests that might be run on it if no one even knows beforehand how many tests you may end up running. For these reasons, it is common practice to convert a many-to-many relationship into two one-to-many relationships, both connected to a new entity that lies between the original two. You can make that conversion with no loss of accuracy, and the problem of how to store things disappears. In Book 2, I go into detail on how to make this conversion.

Simplifying relationships using normalization

Even after you have eliminated all the many-to-many relationships in an ER model, there can still be problems if you have not conceptualized your entities in the simplest way. The next step in the design process is to examine your model and see if adding, changing, or deleting data can cause inconsistencies or even outright wrong information to be retained in your database. Such problems are called *anomalies*, and if there's even a slight chance that they'll crop up, you'll need to adjust your model to eliminate them. This process of model adjustment is called *normalization*, and I cover it in Book 2.

Translating an ER model into a relational model

After you're satisfied that your ER model is not only correct, but economical and robust, the next step is to translate it into a relational model. The relational model is the basis for all relational database management systems. I go through that translation process in Book 2.

- » Seeing where SQL came from
- » Seeing what SQL does
- » Looking at the ISO/IECSQL standard
- » Seeing what SQL doesn't do
- » Examining your SQL implementation options

Chapter 3

Getting to Know SQL

In the early days of relational database management systems (RDBMS), there was no standard language for performing relational operations on data. (If you aren't sure what an RDBMS is, please take a look at the first chapter in this book.) A number of companies came out with relational database management system products, and each had its own associated language. There were some general similarities among the languages because they all performed essentially the same operations on the same kinds of data, structured in the same way. However, differences in syntax and functionality made it impossible for a person using the language of one RDBMS to operate on data that had been stored by another relational database management system. (That's RDBMS, if you missed it the first time.) All the RDBMS vendors tried to gain dominant market share so that their particular proprietary language would prevail. The logic was that once developers learned a language, they would want to stick with it on subsequent projects. This steaming cauldron of ideas set the stage for the emergence of SQL. There was one company (IBM) that had more market power than all the others combined, and it had the additional advantage of being the employer of the inventor of the relational database model.

Where SQL Came From

It is interesting to note that even though Dr. Codd was an IBM employee when he developed the relational database model, IBM's initial support of that model was lukewarm at best. One reason might have been the fact that IBM already had

a leading position in the database market with its IMS (Information Management System) hierarchical DBMS. (For the whole hierarchical versus relational divide, check out Book 1, Chapter 1.) In 1978, IBM released System/38, a minicomputer that came with an RDBMS that was not promoted heavily. As a result, in 1979, the world was introduced to a fully realized RDBMS by a small startup company named Relational Software, Inc. headed by Larry Ellison. Relational's product, called Oracle, is still the leading relational database management system on the market today.

Although Oracle had the initial impact on the market, other companies, including IBM, quickly followed suit. In the process of developing its SQL/DS relational database management system product, IBM created a language, code-named SEQUEL, which was an acronym for Structured English Query Language. This moniker was appropriate because SEQUEL statements looked like English-language sentences, but were more structured than most casual speech.

When it came time for IBM to actually release its RDBMS product, along with its associated language, IBM's legal department flagged a possible copyright issue with the name SEQUEL. In response, management elected to drop the vowels and call the language SQL (pronounced *ess cue el*). The reference to structured English was lost in the process. As a result, many people thought that SQL was an acronym for Structured Query Language. This is not the case. In computer programming, a structured language has some very well-defined characteristics. SQL does not share those characteristics and is thus not a structured language, query or otherwise.

Knowing What SQL Does

SQL is a software tool designed to deal with relational database data. It does far more than just execute queries. Yes, of course you can use it to retrieve the data you want from a database, using a query. However, you can also use SQL to create and destroy databases, as well as modify their structure. In addition, you can add, modify, and delete data with SQL. Even with all that capability, SQL is still considered only a *data sublanguage*, which means that it does not have all the features of general-purpose programming languages such as C, C++, C#, or Java.

SQL is specifically designed for dealing with relational databases, and thus does not include a number of features needed for creating useful application programs. As a result, to create a complete application — one that handles queries as well as provides access to a database — you must write the code in one of the general-purpose languages and embed SQL statements within the program whenever it communicates with the database.

SQL IS A JOKE

There is a joke about the inappropriateness of saying SQL stands for Structured Query Language. It says that SQL is not structured, not restricted to performing queries, and is not a language. The *language* part refers to the fact that SQL is not Turing-complete, as are general-purpose programming languages. (That's Turing as in Alan Turing, code-breaker extraordinaire and one of many godfathers of the computer.) This joke echoes Voltaire's comment about the Holy Roman Empire, which he said was neither holy, Roman, nor an empire.

The ISO/IEC SQL Standard

In the early 1980s, IBM started using SQL in its first relational database product, which was incorporated into the System/38 minicomputer. Smaller companies in the DBMS industry, in an effort to be compatible with IBM's offering, modeled their languages after SQL. In this way, SQL became a de facto standard. In 1986, the de facto standard became a standard de jure when the American National Standards Institute (ANSI) issued the SQL-86 standard. The SQL standard has been continually updated since then, with subsequent revisions named SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2008, SQL:2011, SQL:2016, and most recently, SQL:2023. Along the way, the standard became accepted internationally and became an ISO/IEC standard, where ISO is the International Organization for Standardization, and IEC is the International Electrotechnical Commission. The internationalization of the SQL standard means that database developers all over the world talk to their databases in the same way.

Knowing What SQL Does Not Do

Before I can tell you what SQL doesn't do, I need to give you some background information. In the 1930s, computer scientist and mathematician Alan Turing defined a very simple machine that could perform any computation that could be performed by any computer imaginable, regardless of how big and complex. This simple machine has come to be known as a *universal Turing machine*. Any computer that can be shown to be equivalent to a universal Turing machine is said to be Turing-complete. All modern computers are Turing-complete. Similarly, a computer language capable of expressing any possible computation is said to be Turing-complete. Practically all popular languages, including C, C#, C++, BASIC, FORTRAN, COBOL, Pascal, Java, and many others, are Turing-complete. SQL, however, is not.

Note: Whereas ISO/IEC standard SQL is not Turing-complete, DBMS vendors have added extensions to their versions which *are* Turing complete. Thus the version of SQL that you are working with may or may not be Turing-complete. If it is, you can write a whole program with it, without embedding your SQL code in a program written in another language.

Because standard SQL is not Turing-complete, you cannot write an SQL program to perform a complex series of steps, as you can with a language such as C or Java. On the other hand, languages such as C and Java do not have the data-manipulation facilities that SQL has, so you cannot write a program with them that will efficiently operate on database data. There are several ways to solve this dilemma:

- » Combine the two types of language by embedding SQL statements within a program written in a host language such as C. (I discuss this in Book 5, Chapter 3.)
- » Have the C program make calls to SQL modules to perform data-manipulation functions. (I talk about this in Book 5, Chapter 3 as well.)
- » Create a new language that includes SQL, but also incorporates those structures that would make the language Turing-complete. (This is essentially what Microsoft and Oracle have done with their versions of SQL.)

All three of these solutions are offered by one or another of the DBMS vendors.

Choosing and Using an Available DBMS Implementation

A number of relational database management systems are currently available, and they all include a version of SQL that adheres, more or less, closely to the ISO/IEC international standard for the SQL language. No SQL version available today is completely compliant with the standard, and probably none ever will be. The standard is updated every few years, adding new capability, putting the vendors in the position of always playing catch-up.

In addition, in most cases, the vendors do not *want* to be 100 percent compliant with the standard. They like to include useful features that are not in the standard in order to make their product more attractive to developers. If a developer uses a vendor's nonstandard feature, this has the effect of locking in the developer to that vendor. It makes it difficult for the developer to switch to a different DBMS.

WHAT'S A DATABASE?

There is a lot of confusion about what exactly people are talking about when they mention the word *database*. I have mentioned database management systems (DBMSs), database applications, and databases. Some people lump these things together and call the whole lot databases. Loose usage of this sort just confuses everybody. To keep things clear in your own mind, remember the following distinctions:

- A *database* is a structured collection of integrated records. In other words, it is the data, but organized in a structured way.
- A *database application* is a computer program that operates on a database, which enables users to maintain the database and query it for needed information.
- A *database management system* is the engine that controls access to a database. Database applications must work through a DBMS in order to access the database. Conceptually, the DBMS lies between the database and the database application.

Microsoft Access

Microsoft Access is an entry-level DBMS with which developers can build relatively small and simple databases and database applications. It is designed for use by people with little or no training in database theory. You can build databases and database applications using Access, without ever seeing SQL.

Access does include an implementation of SQL, and you can use it to query your databases — but it is a limited subset of the language, and Microsoft does not encourage its use. Instead, they prefer that you use the graphical database creation and manipulation tools and use the query-by-example (QBE) interface to ask questions of your database. Under the hood and beyond user control, the table-creation tasks that the user specifies using the graphical tools are translated to SQL before being sent to the database engine, which is the part of the DBMS that actually operates on the database.

Microsoft Access runs under any of the Microsoft Windows operating systems, as well as Apple's OS X, but not under Linux or any other non-Microsoft operating system.

To reach the SQL editor in Access, do the following:

1. Open a database that already has tables and at least one query defined.

A great place to start is with the Northwind Traders Starter Edition database provided as a free download with Access. The database includes a built-in