Applied Mathematics for Database Professionals

Lex de Haan and Toon Koppelaars

Apress®

Applied Mathematics for Database Professionals

ISBN-13: 978-1-59059-745-3

ISBN-10: 1-59059-745-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Gennick
Technical Reviewers: Chris Date, Cary Millsap
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jason Gilmore, Jonathan Hassell, Chris Mills, Matthew Moodie, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole Flores
Copy Editor: Susannah Davidson Pfalzer
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Brenda Miller
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/Download section. You will need to answer questions pertaining to this book in order to successfully download the code.

Lex de Haan
1954–2006

"Tall and narrow, with a lot of good stuff on the upper floor"
"A missing value"

Contents at a Glance

Foreword xv
About the Authors xvii
About the Technical Reviewers xix
Acknowledgments xxi
Preface xxiii
Introduction xxv

PART 1 The Mathematics

CHAPTER 1 Logic: Introduction 3
CHAPTER 2 Set Theory: Introduction 23
CHAPTER 3 Some More Logic 47
CHAPTER 4 Relations and Functions 67

PART 2 The Application

CHAPTER 5 Tables and Database States 91
CHAPTER 6 Tuple, Table, and Database Predicates 117
CHAPTER 7 Specifying Database Designs 139
CHAPTER 8 Specifying State Transition Constraints 185
CHAPTER 9 Data Retrieval 199
CHAPTER 10 Data Manipulation 221

PART 3 The Implementation

CHAPTER 11 Implementing Database Designs in Oracle 241
CHAPTER 12 Summary and Conclusions 305

PART 4 Appendixes

APPENDIX A Formal Definition of Example Database 311
APPENDIX B Symbols 333
APPENDIX C Bibliography 335
APPENDIX D Nulls and Three (or More) Valued Logic 337
APPENDIX E Answers to Selected Exercises 347

INDEX 367

Contents

Foreword xv
About the Authors xvii
About the Technical Reviewers xix
Acknowledgments xxi
Preface xxiii
Introduction xxv

PART 1 The Mathematics

CHAPTER 1 Logic: Introduction 3

The History of Logic 4
Values, Variables, and Types 5
Propositions and Predicates 5
Logical Connectives 8
Simple and Compound Predicates 9
Using Parentheses and Operator Precedence Rules 10
Truth Tables 11
Implication 14
Predicate Strength 15
Going a Little Further 15
Functional Completeness 16
Special Predicate Categories 17
Tautologies and Contradictions 17
Modus Ponens and Modus Tollens 18
Logical Equivalences and Rewrite Rules 18
Rewrite Rules 19
Using Existing Rewrite Rules to Prove New Ones 20
Chapter Summary 21
Exercises 22

CHAPTER 2 Set Theory: Introduction 23

Sets and Elements 24
Methods to Specify Sets 26
Enumerative Method 26
Predicative Method 26
Substitutive Method 27
Hybrid Method 27
Venn Diagrams 28
Cardinality and Singleton Sets 29
Singleton Sets 29
The Choose Operator 30
Subsets 30
Union, Intersection, and Difference 31
Properties of Set Operators 33
Set Operators and Disjoint Sets 33
Set Operators and the Empty Set 34
Powersets and Partitions 35
Union of a Set of Sets 36
Partitions 37
Ordered Pairs and Cartesian Product 38
Ordered Pairs 38
Cartesian Product 39
Sum Operator 40
Some Convenient Shorthand Set Notations 41
Chapter Summary 41
Exercises 42

CHAPTER 3 Some More Logic 47

Algebraic Properties 47
Identity 48
Commutativity 48
Associativity 48
Distributivity 49
Reflexivity 49
Transitivity 49
De Morgan Laws 50
Idempotence 50
Double Negation (or Involution) 50
Absorption 50

Quantifiers 51
Quantifiers and Finite Sets 54
Quantification Over the Empty Set 54
Nesting Quantifiers 55
Distributive Properties of Quantifiers 57
Negation of Quantifiers 57
Rewrite Rules with Quantifiers 58
Normal Forms 59
Conjunctive Normal Form 59
Disjunctive Normal Form 60
Finding the Normal Form for a Given Predicate 61
Chapter Summary 63
Exercises 64

CHAPTER 4 Relations and Functions 67

Binary Relations 68
Ordered Pairs and Cartesian Product Revisited 68
Binary Relations 69
Functions 70
Domain and Range of Functions 71
Identity Function 74
Subset of a Function 74
Operations on Functions 75
Union, Intersection, and Difference 75
Limitation of a Function 76
Set Functions 77
Characterizations 78
External Predicates 79
The Generalized Product of a Set Function 79
A Preview of Constraint Specification 81
Function Composition 82
Chapter Summary 84
Exercises 85

PART 2 The Application

CHAPTER 5 Tables and Database States 91

Terminology 91
Database Design 92
Database Variable 92
Database Universe 92
Database State 93
Database 93
Database Management System (DBMS) 93
Table Design 93
Table Structure 93
Table 94
Tables 94
Formal Specification of a Table 94
Shorthand Notation 96
Table Construction 97
Database States 98
Formal Representation of a Database State 98
Database Skeleton 100
Operations on Tables 101
Union, Intersection, and Difference 101
Restriction 104
Join 106
Attribute Renaming 108
Extension 109
Aggregation 111
Chapter Summary 113
Exercises 114

CHAPTER 6 Tuple, Table, and Database Predicates 117

Tuple Predicates 118
Table Predicates 120
Database Predicates 124
A Few Remarks on Data Integrity Predicates 127

Common Patterns of Table and Database Predicates 127
Unique Identification Predicate 127
Subset Requirement Predicate 129
Specialization Predicate 132
Generalization 134
Tuple-in-Join Predicate 135
Chapter Summary 137
Exercises 138

CHAPTER 7 Specifying Database Designs 139

Documenting Databases and Constraints 140
The Layers Inside a Database Design 141
Top-Down View of a Database 141
Classification Schema for Constraints 142
Specifying the Example Database Design 143
Database Skeleton 144
Characterizations 147
Tuple Universes 151
Table Universes 156
Database Universe 167
Chapter Summary 182
Exercises 183

CHAPTER 8 Specifying State Transition Constraints 185

More Data Integrity Predicates 185
A Simple Example 186
State Transition Predicates 188
State Transition Constraints 190
State Transition Universe 190
Completing the Example Database Design 192
Chapter Summary 196
Exercises 197

CHAPTER 9 Data Retrieval 199

Formally Specifying Queries 199
Example Queries Over DB_UEX 201
A Remark on Negations 216
Chapter Summary 218
Exercises 219

CHAPTER 10 Data Manipulation 221

Formally Specifying Transactions 221
Example Transactions Over DB_UEX 226
Chapter Summary 236
Exercises 236

PART 3 The Implementation

CHAPTER 11 Implementing Database Designs in Oracle 241

Introduction 242
Window-on-Data Applications 243
Classifying Window-on-Data Application Code 243
Implementing Data Integrity Code 247
Alternative Implementation Strategies 248
Order of Preference 254
Implementing Table Structures 255
Implementing Attribute Constraints 259
Implementing Tuple Constraints 262
Table Constraint Implementation Issues 265
DI Code Execution Models 266
DI Code Serialization 284
Implementing Table Constraints 290
Implementing Database Constraints 291
Implementing Transition Constraints 296
Bringing Deferred Checking into the Picture 299
Why Deferred Checking? 299
Outline of Execution Model for Deferred Checking 300
The RuleGen Framework 303
Chapter Summary 304

CHAPTER 12 Summary and Conclusions 305

Summary 305
Conclusions 306

PART 4 Appendixes

APPENDIX A Formal Definition of Example Database 311

Bird's Eye Overview 312
Database Skeleton DB_S 313
Table Universe Definitions 314
Some Convenient Sets 315
Table Universe for EMP 315
Table Universe for SREP 317
Table Universe for MEMP 317
Table Universe for TERM 318
Table Universe for DEPT 319
Table Universe for GRD 319
Table Universe for CRS 321
Table Universe for OFFR 321
Table Universe for REG 322
Table Universe for HIST 323
Database Characterization DBCH 324
Database Universe DB_UEX 325
State Transition Universe TX_UEX 330

APPENDIX B Symbols 333

APPENDIX C Bibliography 335

Original Writings That Introduce the Methodology Demonstrated in This Book 335
Recommended Reading in the Area of the Underlying Mathematical Theories 335
Seminal Writings That Introduce the General Theory of Data Management 335
Recommended Reading on Relational Database Management 335
Research Papers on Implementing Data Integrity Constraints and Related Subjects 336
Previous Related Writings of the Authors 336

APPENDIX D Nulls and Three (or More) Valued Logic 337

To Be Applicable or Not 337
Inapplicable. 338
Not Yet Applicable 338
Nice to Know. 339
Implementation Guidelines. 340
Three (or More) Valued Logic 340
Unknown 340
Truth Tables of Three-Valued Logic 341
Missing Operators 343
Three-Valued Logic, Tautologies, and Rewrite Rules 343
Handling Three-Valued Logic. 344
Four-Valued Logic 345

APPENDIX E Answers to Selected Exercises 347

Chapter 1 Answers 347
Chapter 2 Answers 352
Chapter 3 Answers 353
Chapter 4 Answers 355
Chapter 5 Answers 356
Chapter 6 Answers 357
Chapter 7 Answers 358
Chapter 8 Answers 360
Chapter 9 Answers 361
Chapter 10 Answers 364

INDEX 367

Foreword

We welcome this contribution to the database literature. It is another book on the theory and practice of relational databases, but this one is interestingly different. The bulk of the book is devoted to a treatment of the theory. The treatment is not only rigorous and mathematical, but also rather more approachable than some other texts of this kind. The authors clearly recognize, as we do, the importance of logic and mathematics if database study is to be taken seriously. They have done a good job of describing a certain formalism developed by their former teachers, Bert de Brock and Frans Remmen. This formalism includes some ideas that will be novel to many readers, even those who already have a degree of familiarity with the subject. A particularly interesting novel idea, to us, is the formalization of updating and transactions in Chapter 10.

The formalism is interestingly different from the approach we and several others have adopted. The differences do not appear to be earth-shattering, but the full consequences are not immediately obvious to us, and we hope they will provoke discussion. One major difference is the omission of any in-depth treatment of types. This omission can be justified because relations are orthogonal to the types available for their attributes, so the two issues are somewhat separable even though any relational database language must of course deal with both of them. Closely related to the issue of types, and possibly a consequence of the omission we have noted, is the fact that there is only one empty relation under the De Brock/Remmen approach; by contrast, we subscribe to the notion that empty relations of distinct types are themselves distinct. Distinguishing empty relations of different types is in any case needed for static type checking, regarded as a *sine qua non* for database languages intended for use by robust applications. However, our approach is explicitly intended to provide, among other things, a foundation for database language design. The De Brock/Remmen approach appears to be more focused on getting the specifications of the database and its transactions right.

The authors turn from theory to practice in Chapter 11. Here they deal exclusively with *current* practice, showing how to translate the theoretical solutions described in Part 2 into concrete syntax using a particular well-known implementation of SQL. In doing this they have to face up not only to SQL's well-known deviations from the theory (especially its deviations from the classical logic used in that theory) but also to some alarming further deficiencies in many of the implementations of that language. This chapter can therefore be read as a thinly veiled wake-up call to DBMS vendors. We would like to take this opportunity to lift that veil a little.

The two deficiencies brought out by the authors' example solutions are (a) severe limitations on the extent to which declarative database constraints can be expressed, and (b) lack of proper support for serializability. If a DBMS does not suffer from (a), then (b) comes into play only when declarative constraints would lead to unacceptable performance. In that case, custom-written procedures, preferably triggered procedures, might be needed as a

workaround. Such procedures require very careful design, as the authors eloquently demonstrate, so they are prone to some of those very errors that the relational approach from its very beginning was expressly designed to avoid. Deficiency (b) compounds this exposure by requiring the developers of these procedures to devise and implement some form of concurrency control within them. The authors describe a method requiring them first to devise a locking scheme, appropriate to the particular requirements of the database, then inside the procedures, in accordance with that scheme, to acquire locks at appropriate points. If the DBMS suffers from deficiency (a), then procedural workarounds will be needed for many more constraints, regardless of performance considerations. The worst-case scenario is clearly when the DBMS suffers from both (a) and (b). That is the scenario the authors feel compelled to assume, given the current state of the technology. To our minds, a relational DBMS that suffers from (a) is not a relational DBMS. (Of course, we have other reasons for claiming that no SQL DBMS is a relational DBMS anyway.) A DBMS that suffers from (b) is not a DBMS.

Standard SQL is relationally complete in its support for declarative constraints by permitting the inclusion of query expressions in the `CHECK` clause of a constraint declaration. However, it appears that few SQL products actually support that particular standard feature. The authors offer a polite excuse for this state of affairs. The excuse is well understood. It goes like this: a constraint that includes a possibly complex query against a possibly very large database might take ages to evaluate and might require frequent evaluation in the presence of a high transaction rate. We do not yet know—and it is an important and interesting research topic—how to do the kind of optimization that would be needed for the DBMS to work out efficient evaluation strategies along the lines of the authors' custom-written solutions. Performance considerations would currently militate against big businesses with very large databases and high transaction rates expressing the problematical constraints declaratively. Such businesses would employ—and could afford to employ—software experts to develop the required custom-written procedures. But what about small businesses with small databases and low transaction rates? And what about constraints that are evaluated quickly enough in spite of the inclusion of a query expression (which might be nothing more than a simple existence test, for example)? In any case, our history is littered with good ideas (for example, FORTRAN in the 1960s, the relational model in the 1970s) that have initially been shunned on advice from the performance sages that has eventually turned out to be ill-founded. As the years go by machines get faster, memory gets bigger, research comes up with new solutions. So much, then, for that excuse.

Sadly, Lex de Haan did not live to see the completion of this joint project. We would like to express our appreciation to Toon Koppelaars for the work that he has undertaken single-handedly since the untimely death in early 2006 of his friend and ours.

Hugh Darwen and Chris Date

About the Authors

LEX DE HAAN studied applied mathematics at the University of Technology in Delft, the Netherlands. His experience with Oracle goes back to the mid 1980s, version 4. He worked for Oracle Corp. from 1990 until 2004 in various education-related roles, ending up in Server Technologies (Oracle product development) as senior curriculum manager for the advanced DBA curriculum group. In that role, he was involved in the development of Oracle9*i* and Oracle Database 10*g*. In March 2004, he decided to go independent and founded his own company, Natural Join B.V. (`http://www.naturaljoin.nl`). From 1999 until his passing in 2006, he was involved in the ISO SQL language standardization process as a member of the Dutch national body. He was also one of the founding members of the OakTable network (`http://www.oaktable.net`). He wrote the well-received *Mastering Oracle SQL and SQL*Plus* (Apress, 2005).

TOON KOPPELAARS studied computer science at the University of Technology in Eindhoven, the Netherlands. He is a longtime Oracle technology user, having used the Oracle database and tool software since 1987. During his career he has been involved in application development (terminal/host in the early days, GUI client/server later on, and J2EE/web development nowadays), as well as database administration. His interest areas include performance tuning (ensuring scalability and SQL tuning), architecting applications in a database-centric way, and database design. Within the database design area, the mathematical specification and robust implementation of database designs—that is, including the data integrity constraints (often referred to as business rules)—is one of his special interest areas. He is employed as a senior IT architect at Centraal Boekhuis B.V., a well-known Oracle shop in the Netherlands. Toon is also a frequent presenter at Oracle-related conferences.

About the Technical Reviewers

CHRIS DATE is an independent author, lecturer, researcher, and consultant specializing in relational database systems. He was one of the first people anywhere to recognize the fundamental importance of Ted Codd's pioneering work on the relational model. He was also involved in technical planning for the IBM products SQL/DS and DB2 at the IBM Santa Teresa Laboratory in San Jose, California. He is best known for his books—in particular *An Introduction to Database Systems*, Eighth Edition (Addison-Wesley, 2003), the standard text in the field, which has sold nearly three quarters of a million copies worldwide—and (with Hugh Darwen and Nikos A. Lorentzos) *Temporal Data and the Relational Model* (Morgan Kaufmann, 2002).

CARY MILLSAP is the principal author of *Optimizing Oracle Performance* (O'Reilly, 2003) and the lead designer and developer of the Hotsos PD101 course. Prior to cofounding Hotsos in 1999, he served for ten years at Oracle Corp. as one of the company's leading system performance experts. At Oracle, he also founded and served as vice president of the 80-person System Performance Group. He has educated thousands of Oracle consultants, support analysts, developers, and customers in the optimal use of Oracle technology through his commitment to writing, teaching, and speaking at public events.

Acknowledgments

This project started at the end of the summer in 2005, and due to unfortunate circumstances took much longer than originally planned. My coauthor was already diagnosed when he contacted me to jointly write a book "about the mathematics in our profession." We started writing in October 2005 and first spent a lot of time together developing the example database design—especially the involved data integrity constraints—that would be used throughout the book. "I'll start at the beginning, you start at the end, then we will meet somewhere in the middle," is what he said once the database design was finished. In the months that followed Lex put in a big effort to create the Introduction through Chapter 3. Unfortunately, around Christmas 2005 his situation deteriorated rapidly and he never saw the rest of this book.

Thankfully, I received the support of many other people to complete this project.

The contribution of my main reviewers to the quality of this book has been immense. I'd like to thank Chris Date, Cary Millsap, Hugh Darwen, and Frans Remmen for their efforts in reviewing the manuscript and offering me many comments and suggestions to improve the book.

I also greatly appreciate the time some of my colleagues at Centraal Boekhuis put into this book: Jaap de Klerk, Lotte van den Hoek, and Emiel van Bockel for reviewing several chapters; Petra van der Craats for helping me out when I was struggling with the English language; and Ronald Janssen for his overall support for this project.

I must mention the support I received from the people at Apress, especially my editor Jonathan Gennick for remaining committed to this book even when the original plan had to be completely revised, and the other people at Apress involved in the production of this book—the only ones I've never met in person: Tracy Brown Collins, Tina Nielsen, Susannah Davidson Pfalzer, Kelly Winquist, and April Eddy.

I would like to mention a few other people: Mogens Nørgaard, for his support early in 2006 in the time right after Lex's passing; Juliette Nuijten, for her continued support of this project; and Bert de Brock, for the influential curriculum on database systems he provided together with Frans Remmen more than 20 years ago. Without those courses this book would not exist.

I must of course also thank my wife for giving me the opportunity to embark upon this project and to finish it, and our kids for their patience, especially these last three months; I had to promise them not even to think about writing another book.

And finally, I thank Lex, who initiated this book and decided to contact me as his coauthor.

Well Lex, we finally did it, your last project is done.

Toon Koppelaars
Zaltbommel, the Netherlands, March 2007

Preface

This book is not an easy read, but you will have a great understanding of mathematics as it relates to database design by the end.

We'll introduce you to a mathematical methodology that enables you to deal with database designs in a clear way. Those of you who have had a formal education in mathematics will most likely enjoy reading this book. It demonstrates how you can apply two mathematical disciplines—logic and set theory—to your current profession.

For those of you who lack a formal education in mathematics, you'll have to put in an effort when reading this book. We do our best to explain all material in a clear way and provide sufficient examples along the way. Nevertheless, there is a lot of new material for you to digest.

We assume that you are familiar with designing a database. This book will not teach you how to design databases; more specifically, this book will not explain what makes a database design a good one or a bad one. This book's primary goal is to teach you a formal methodology for specifying a database design; in particular, for specifying all involved data integrity constraints in a clear and unambiguous manner.

This book is a *must* for every IT professional who is involved in any way with designing databases:

- Database designers, data architects, and data administrators
- Application developers with database design responsibilities
- Database administrators with database design responsibilities
- IT architects
- People managing teams that include any of the preceding roles

We wrote this book because we are convinced that the mode of thought required by this formal methodology will—as an important side effect—contribute to your database design capabilities. Understanding this formal methodology will benefit you, the database professional, and will in the end make you a better database designer.

Introduction

This book will not try to change your attitude towards mathematics, which can be anywhere between hate and love. The sole objective of this book is to show you how you can use mathematics in your life as a database professional, and how mathematics can help you solve certain problems. We, the authors, are convinced that familiarity with the areas of mathematics that will be presented in this book, and on which *the relational model of data* is based, is a strong prerequisite for anybody who aims to be professionally involved with databases.

This book tries to fill a space that is not yet covered by the many books on databases that are already available. In Part 1, we cover just the part of mathematics that is useful for the practice of the database professional; the mathematical theory covered in this part is linked to the practice in Parts 2 (specifying database designs) and 3 (implementing database designs).

One thing is for sure: mathematics forces you to think clearly and precisely, and then to write things down as formally and precisely as possible. This is because the language of mathematics is both formal and rich in expressive power. Natural languages are rich in expressive power but are highly informal; on the other hand, programming languages are formal but typically have much less expressive power than mathematics.

Mathematicians

Mathematicians are strange people. Most of them have all sorts of weird hobbies, and they all share a passionate love for puzzles and games. To be more precise, they love to create their own games and then play those games. Well, how do you create a game? You simply establish a set of rules, and start playing. If you are the creator of the game, you have a rather luxurious position: if you don't like the game that much, you simply revisit the rules, implement some changes, and start playing again—until you like the game.

Mathematicians always strive for elegance and *orthogonality*—they dislike exceptions. A game is said to be designed in an orthogonal way if its set of components that together make up the whole game capability are non-overlapping and mutually independent. Each capability should be implemented by only one component, and one component should only implement one capability of the game. Well-separated and independent components ensure that there are no side effects: using or even changing one component does not cause side effects in another area of the game.

Note For more information, see for example "A Note on Orthogonality" by C. J. Date, originally published in *Database Programming & Design* (July 1995), or visit `http://en.wikipedia.org/wiki/Orthogonality`.

Why do things in a complicated way if you can accomplish the same thing in a more simple way? Why allow tricks in certain places, but at the same time forbid them in other places where the same trick would make a lot of sense? Exceptions are the worst of all. Therefore, mathematicians always explore the boundaries of their games. If the established rules don't behave nicely at the boundaries, there is room for improvement.

High-Level Book Overview

Over time, mathematicians have spawned several formal disciplines. This book pays special attention to the following two formal disciplines, because they are the most relevant ones in the application of mathematics to the field of databases:

- Logic
- Set theory

The first part of this book consists of four chapters; they introduce the mathematics as such. While reading these chapters, you should try to exercise some patience in case you don't immediately see their relevance for you; they lay down the mathematical concepts, techniques, and notations needed for the second and third parts of the book.

Note Even if you think at first glance that your mathematical skills are strong enough, we advise you to read and study the first four chapters in detail and to go through all exercises, without looking at the corresponding solutions first. This will help you get used to the mathematical notations used throughout this book; moreover, some exercises are designed to make you aware of certain common errors.

The second part consists of Chapters 5 through 10, showing the application of the mathematics to database issues. Chapter 5 introduces a formal way to specify table designs and introduces the concept of a database state. Chapter 6 establishes the notion of data integrity predicates; we use these to specify data integrity constraints. Chapter 7 specifies a full-fledged example database design in a clear mathematical form. You'll discover through this example that specifying a database design involves specifying data integrity constraints for the most part. Chapter 8 adds the notion of state transition constraints, and formally specifies these for the given example database design. Chapter 9 shows how you can precisely formulate queries in mathematics, and Chapter 10 shows how you can formally specify transactions.

The third part consists of Chapter 11 and Chapter 12. Chapter 11 goes into the details of realizing a database design, especially its data integrity constraints, in a database management system (DBMS)—a crucial and challenging task for any database professional. In Chapter 11, we establish a further link from the theory to the SQL DBMS practice of today.

Note Chapter 11 is an optional chapter. However, if you're involved in implementing database designs in Oracle's SQL DBMS, you'll appreciate it.

Chapter 12 summarizes the book, lists some conclusions, and provides some general guidelines.

The book contains several appendixes:

- Appendix A gives the full formal definition of the database design used in the book.
- Appendix B contains a quick reference of all mathematical symbols used in the book.
- Appendix C provides a reference for further background reading.
- Appendix D provides a brief exploration of the use of `NULL`s.
- Appendix E provides solutions for selected exercises.

We assume that you're aware of the existence of the relational model, and perhaps you also have some in-depth knowledge of what this model is about (that's not required, though). We also assume that you have experience in designing databases, and you're therefore familiar with concepts such as keys, foreign keys, (functional) dependencies, and the third normal form (the latter two aren't required).

This book's main focus is on specifying a relational database design in general and specifying the data integrity constraints involved in such a design, specifically. We demonstrate how elementary set theory (in combination with logic) aids us in producing solid database design specifications that give us a good and clear insight into the relevant constraints.

Other authors, most notably C. J. Date in his recent book *Database In Depth* (O'Reilly, 2005), lay out the fundamentals of the relational model but sometimes assume you are knowledgeable in certain mathematical disciplines. In this book no mathematical knowledge is preassumed; we'll deliver the theoretical—set-theory—concepts that are necessary to define a relational database design from the ground upwards.

We must mention up front that the approach taken in this book is a different approach (for some, maybe radically different) to the one taken by other authors. The methodology that is developed in this book uses merely elementary set theory in conjunction with logic. Elementary set theory suffices to specify relational database designs, including all relevant data integrity constraints. We'll also use set theory as the vehicle to specify queries and transactions on such designs.

Note We (the authors) are not the inventors of the methodology presented in this book. Frans Remmen and Bert de Brock originally developed this methodology in the 1980s, while they were both engaged at the Eindhoven University of Technology. Appendix C lists two references of books authored by Bert de Brock in which he introduces this methodology to specify database designs.

Database Design Implementation Issues

The majority of all DBMSes these days are based on the ISO standard of the SQL (pronounced as "ess-cue-ell") language. This is where you'll get into some trouble. First of all, the SQL language is far from an elegant and orthogonal database language; furthermore, it is not too difficult to see that it is a product of years of political debates and attempts to achieve consensus.

In hindsight, some battles were won by the wrong guys. Indeed, this is one of the reasons why C. J. Date and Hugh Darwen wrote their book on what they call "the third manifesto." A fully revised third edition was published in 2006: *Databases, Types, and the Relational Model: The Third Manifesto* (Addison-Wesley).

On top of this, several database software vendors have made mistakes—sometimes small ones, sometimes big ones—in their attempts to implement the ISO standard in their products. They also left certain features out and added nonstandard features to enrich their products, thus deviating from the ISO standard. As soon as you try to step away from mathematics (and thus from the relational model) and start using an SQL DBMS, you'll inevitably open up several cans of worms.

This book tries to stay away as much as possible from SQL, thus keeping the book as generic as possible. Chapters 9 (data retrieval) and 10 (data manipulation) display SQL expressions; they serve only to demonstrate (in)abilities of this language in comparison to the mathematical formalism introduced in this book. Both authors happen to have extensive experience with the SQL DBMS from Oracle; the SQL code given in these chapters is compliant with the 10*g* release of Oracle's SQL DBMS. Chapter 11 (implementing database designs) displays SQL expressions even more. We'll also maintain the Oracle-specific content of Chapter 11; you can download the code from the Source Code/Download area of the Apress Web site (`http://www.apress.com`).

PART 1

The Mathematics

Everything should be made as simple as possible, but not simpler.

Albert Einstein (1879–1955)

CHAPTER 1

Logic: Introduction

The word "logic" has many meanings, and is heavily overloaded. It's derived from the Greek word *logicos*, meaning "concerning language and speech" or "human reasoning."

The section "The History of Logic" gives a concise overview of the history of logic, just to show that many brilliant people have been involved over several centuries to develop what's now known as mathematical logic. The section is by no means meant to be complete.

In the section "Values, Variables, and Types," we'll discuss the difference between *values* (constants) and *variables*. We'll also introduce the notion of the *type of a variable*. We'll use these notions in the section "Propositions and Predicates" to introduce the concept of a *predicate*, and its special case, a *proposition*—the main concepts in logic.

The section "Logical Connectives" explains how you can build new predicates by combining existing predicates using *logical connectives*. Then, in the section "Truth Tables" you'll see how you can use *truth tables* to define logical connectives and to investigate the truth value of logical expressions. Truth tables are an important and useful tool to start developing various concepts in logic.

Functional completeness is covered in the section "Functional Completeness"; it's about which logical connectives you need (as a minimum) to formulate all possible logical expressions.

The following two sections introduce the concepts of *tautologies* and *contradictions*, *logical equivalence*, and *rewrite rules*. You can use a rewrite rule to transform one logical expression into another (equivalent) logical expression.

This chapter is an introductory chapter on logic. Chapter 3 will continue where this one stops—the two chapters make up one single topic (logic). The split is necessary because some concepts concerning logic require the introduction of a few set-theory notions first. Chapter 2 will serve that purpose.

The introduction of the crucial concept of *rewrite rules* at the end of this chapter opens up the first possibility to do some useful exercises. They serve two important purposes:

- Learning how to use truth tables and rewrite rules to investigate logical expressions
- Getting used to the mathematical symbols introduced in this chapter

Therefore, we strongly advise you to spend sufficient time on these exercises before moving on to other chapters.

The History of Logic

The science of logic and the investigation of human reasoning goes back to the ancient Greeks, more than 2,000 years ago. Aristotle (384–322 BC), a student of Plato, is commonly considered the first logician.

Gottfried Wilhelm Leibnitz (1646–1716) established the foundations for the development of mathematical logic. He thought that symbols were extremely important to understand things, so he tried to design a universal symbolic language to describe human reasoning. The logic of Leibnitz was based on the following two principles:

- There are only a few simple ideas that form the "alphabet of human thought."
- You can generate complex ideas from those simple ideas by a process analogous to arithmetical multiplication.

George Boole (1815–1864) invented the general concept of a Boolean algebra, the foundation of modern computer arithmetic. In 1854 he published *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities*, in which he shows that you can perform arithmetic on logical symbols just like algebraic symbols and numbers.

In 1922, Ludwig Wittgenstein (1889–1951) introduced truth tables as we know them today, based on the earlier work of Gottlob Frege (1848–1925) and others during the 1880s.

After a formal notation was introduced, several attempts were made to use mathematical logic to describe the foundation of mathematics itself. The attempt by Gottlob Frege (that failed) is famous; Bertrand Russell (1872–1970) showed in 1901 that Frege's system could produce a contradiction: the famous Russell's paradox (see sidebar). Later attempts to achieve the same goal were performed by Bertrand Russell and Alfred North Whitehead (1861–1947), David Hilbert (1862–1943), John von Neumann (1903–1957), Kurt Gödel (1906–1978), and Alfred Tarski (1902–1983), just to name a few of the most famous ones.

RUSSELL'S PARADOX

Russell's paradox can be difficult to understand for readers who are unfamiliar with mathematical logic in general and with setting up a mathematical proof in particular. The paradox goes as follows:

1. Consider the set of all sets that are not members of themselves; let's call this set X.
2. Suppose X is an element of X—but then it must *not* be a member of itself, according to the preceding definition of set X. So the supposition is FALSE.
3. Similarly, suppose X is *not* an element of X—but then it must be a member of itself, again according to the preceding definition of set X. So this supposition is FALSE too.
4. But surely one of these two suppositions must be TRUE; hence the paradox.

Don't worry if this puzzles you; it isn't important for the application of the mathematics that this book deals with.

It's safe to say that the science of logic is sound; it has existed for many centuries and has been investigated by many brilliant scientists over those centuries.

■ **Note** If you want to know more about the history of logic, or the history of mathematics in general, `http://en.wikipedia.org` is an excellent source of information.

These days, formal methods derived from logic are not only used in mathematics, informatics (computer science), and artificial intelligence; they are also used in biology, linguistics, and even in jurisprudence.

The importance of data management being based on logic was envisioned by E. F. (Ted) Codd (1923–2003) in 1969, when he first proposed his famous relational model in the IBM research report "Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks." The relational model for database management introduced in this research report has proven to be an influential general theory of data management and remains his most memorable achievement.

Every science is (should be) based on logic: every theory is a system of sentences (or statements) that are accepted as true and can be used to *derive* new statements following some well-defined rules. It's one of the main goals of this book to explain the mathematical concepts on which the science of relational data management is based.

Values, Variables, and Types

You probably have some idea of the two terms *values* and *variables*. However, it's important to define these two terms precisely, because they are often misunderstood and mixed up.

A *value* is an individual constant with a well-determined meaning. For example, the integer `42` is a value. You cannot update a value; if you could, it would no longer be the same value. Values can be represented in many ways, using some encoding. Values can have any arbitrary complexity.

A *variable* is a holder for a value. Variables in the course of time and space get a different value. We call the changing of values the *updating* of a variable. Variables have a *name*, allowing you to talk about them without knowing which value they currently represent.

In this book, we'll always use variables that are of a given *type*. The *set of values* from which the variable is allowed to hold a value is referred to as the *type of that variable*.

A database—containing table values—is an example of a variable; at any point in time, it has a certain (complex) value. *We will specify the type of a database variable in a precise way in Part 2 of this book.*

Propositions and Predicates

In logic, the main components we deal with are propositions and predicates. A proposition is a *declarative sentence* that's either `TRUE` or `FALSE`.

■ **Note** A sentence S is a declarative sentence, if the following is a proper English question: "Is it true that S?"

If a proposition is true we say it has a "truth value" of TRUE; if a proposition is false, its truth value is FALSE. Here are some examples of propositions:

- This book has two authors.
- The square root of 16 equals 3.
- All mathematicians are liars.
- If Toon is familiar with SQL, then Lex has three daughters.

All four examples are declarative sentences; if you prefix them with "Is it true that," then a proper English question is formed and you can decide if the declarative sentences are TRUE or not. The truth value of the first proposition is TRUE; it is a TRUE proposition. The second proposition is obviously FALSE; the square root of 16 equals 4. The third example is a proposition too, although you might find it difficult to decide what its truth value is. But in theory you *can* decide its truth value. Assuming you have a clear definition of who is and who isn't a mathematician, you can determine the set of persons that need to be checked. You would then have to check every mathematician, in this rather large but finite set, to find the truth value of the proposition. If you find a non-lying mathematician, then the proposition is FALSE; on the other hand, if no such mathematician can be found, then the proposition is clearly TRUE. In the last example, "Toon" and "Lex" are the authors of this book. Therefore, you should be able to decide whether this predicate is TRUE or FALSE if you know enough about the authors of this book. The proposition is FALSE, by the way; Toon is indeed familiar with SQL, but Lex does not have three daughters.

■ **Note** It's a common misconception to consider only TRUE statements to be propositions; propositions can be FALSE as well.

The following examples are not propositions:

- `x + y > 10`
- The square root of x equals z.
- What did you pay for this book?
- Stop designing databases.

The first two examples hold embedded variables; the truth value of these sentences depends on the values that are currently held by these variables. The last two examples aren't declarative sentences and therefore aren't propositions.

A *predicate* is something having the form of a declarative sentence. A predicate holds embedded variables whose values are unknown at this time; you cannot decide if what is declared is either TRUE or FALSE without knowing the value(s) for the variable(s). We'll refer to the embedded variables in a predicate as the *parameters* of the predicate.

The first two examples in the preceding list are predicates; they hold embedded variables x, y, and z. Following are some other examples of predicates:

- i has the value 4.
- x lives in y.
- If Toon is familiar with SQL, then Lex has k daughters.

You cannot tell if the first example is TRUE or FALSE, because it depends on the actual value of parameter i. The same holds for the second example; as long as you don't know which human being is represented by parameter x and which city is represented by parameter y, you cannot say whether this predicate is TRUE or FALSE. Finally, the truth value of the last example depends on its parameter k. You already saw that if value 3 is substituted for parameter k, then the truth value of this predicate becomes FALSE.

A predicate with n parameters is called an n-place predicate. If you substitute one of the parameters in an n-place predicate with a value, then the predicate becomes an (n-1)-place predicate. For instance, the preceding second example is a 2-place predicate; it has two parameters, x and y. If you substitute the value Lex for parameter x, then this predicate turns into the following expression:

```
Lex lives in y
```

This expression represents a 1-place predicate; it has one parameter. The truth value still depends upon (the value of) the remaining parameter. If you now substitute value "Utrecht"—the name of a city in the Netherlands—for parameter y, the predicate turns into a 0-place predicate.

```
Lex lives in Utrecht
```

Do you see that this is now a proposition? You can decide the truth value of this expression. As this example shows, propositions can be regarded as a special case of predicates; they are predicates with no parameters. You can convert a predicate into a proposition by providing values that are substituted for the parameters. This is called *instantiating* the predicate with the given values.

■ **Note** A way to look at a predicate is as follows—here we quote from Chris Date's book *Database in Depth* (O'Reilly Media, 2005): "You can think of a predicate as a truth-valued function. Like all functions, it has a set of parameters, it returns a result when it is invoked (instantiated) by supplying values for the parameters, and, because it's truth valued, that result is either TRUE or FALSE."

The parameters of a predicate are also referred to as the *free variables* of a predicate. There's another way to convert predicates into propositions: by *binding* the involved free

variable(s) with a *quantifier*. Free variables then turn into what are called *bound variables*. Quantification (over a set) is an important concept in logic and even more so in data management; we'll cover it in Chapter 3 (the section "Quantifiers") in great detail.

Table 1-1 summarizes the properties of a predicate and a proposition.

Table 1-1. *Predicates and Propositions*

Predicate	Proposition
Form of declarative sentence	Declarative sentence
With parameters	Without parameters
Truth-valued function	Either `TRUE` or `FALSE`
Input is required to evaluate truth/falsehood	Special case predicate (no input required)

Before we go on, let's briefly look at sentences such as the following:

```
This statement is false
I am lying
```

These are *self-referential* sentences; they say something about themselves. Sentences of this type can cause trouble in the sense that they might lead to a contradiction. The second example is known as the liar's paradox. If you assume these sentences to be `TRUE` then you can draw the conclusion that they are `FALSE` (and vice versa); you cannot decide whether they are `TRUE` or `FALSE`, which is a mandatory property for them to be valid propositions. The solution is simply to disqualify them as valid propositions.

You must also discard "ill-formed" expressions as valid predicates; that is, expressions that don't adhere to our syntax rules, such as the following:

```
3 is an element of 4
n = 4 ∨ ∧ m = 5
```

The first one is ill-formed because `4` is not a set; it is a numeric value. To say `3` is an element of it doesn't make any sense. Here we assume you have some idea of the concept of a set and for something to be an element of a set; we'll cover elementary set theory in Chapter 2. The second expression contains two consecutive connectives ∨ and ∧ (explained in the next section), which is illegal.

Logical Connectives

You can build new predicates by applying *logical connectives* to existing ones. The most well-known connectives are *conjunction* (logical `AND`), *disjunction* (logical `OR`), and *negation* (logical `NOT`); two other connectives we'll use frequently are *implication* and *equivalence*. Throughout this book, we use mathematical symbols to denote these logical connectives, as shown in Table 1-2.