

DATABASE SYSTEMS

The Complete Book

DATABASE SYSTEMS

The Complete Book

Second Edition

Hector Garcia-Molina

Jeffrey D. Ullman

Jennifer Widom

*Department of Computer Science
Stanford University*



Upper Saddle River, New Jersey 07458



NOTICE:

This work is protected by U.S. copyright laws and is provided solely for the use of college instructors in reviewing course materials for classroom use. Dissemination or sale of this work, or any part (including on the World Wide Web), is not permitted.

Editorial Director, Computer Science and Engineering: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Editorial Assistant: *Melinda Haggerty*
Director of Marketing: *Margaret Waples*
Marketing Manager: *Christopher Kelly*
Senior Managing Editor: *Scott Disanno*
Production Editor: *Irwin Zucker*
Art Director: *Jayne Conte*
Cover Designer: *Margaret Kenselaar*
Cover Art: *Tamara L. Newman*
Manufacturing Buyer: *Lisa McDowell*
Manufacturing Manager: *Alan Fischer*



© 2009, 2002 by Pearson Education Inc.
Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, NJ 07458

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Pearson Prentice Hall™ is a trademark of Pearson Education, Inc.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-606701-8

978-0-13-606701-6

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*
Pearson Educación de México, S.A. de C.V.
Pearson Education—Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

Preface

This book covers the core of the material taught in the database sequence at Stanford. The introductory course, CS145, uses the first twelve chapters, and is designed for all students — those who want to use database systems as well as those who want to get involved in database implementation. The second course, CS245 on database implementation, covers most of the rest of the book. However, some material is covered in more detail in special topics courses. These include CS346 (implementation project), which concentrates on query optimization as in Chapters 15 and 16. Also, CS345A, on data mining and Web mining, covers the material in the last two chapters.

What's New in the Second Edition

After a brief introduction in Chapter 1, we cover relational modeling in Chapters 2–4. Chapter 4 is devoted to high-level modeling. There, in addition to the E/R model, we now cover UML (Unified Modeling Language). We also have moved to Chapter 4 a shorter version of the material on ODL, treating it as a design language for relational database schemas.

The material on functional and multivalued dependencies has been modified and remains in Chapter 3. We have changed our viewpoint, so that a functional dependency is assumed to have a set of attributes on the right. We have also given explicitly certain algorithms, including the “chase,” that allow us to manipulate dependencies. We have augmented our discussion of third normal form to include the 3NF synthesis algorithm and to make clear what the tradeoff between 3NF and BCNF is.

Chapter 5 contains the coverage of relational algebra from the previous edition, and is joined by (part of) the treatment of Datalog from the old Chapter 10. The discussion of recursion in Datalog is either moved to the book's Web site or combined with the treatment of recursive SQL in Chapter 10 of this edition.

Chapters 6–10 are devoted to aspects of SQL programming, and they represent a reorganization and augmentation of the earlier book's Chapters 6, 7, 8, and parts of 10. The material on views and indexes has been moved to its own chapter, number 8, and this material has been augmented with a discussion of

important new topics, including materialized views, and automatic selection of indexes.

The new Chapter 9 is based on the old Chapter 8 (embedded SQL). It is introduced by a new section on 3-tier architecture. It also includes an expanded discussion of JDBC and new coverage of PHP.

Chapter 10 collects a number of advanced SQL topics. The discussion of authorization from the old Chapter 8 has been moved here, as has the discussion of recursive SQL from the old Chapter 10. Data cubes, from the old Chapter 20, are now covered here. The rest of the chapter is devoted to the nested-relation model (from the old Chapter 4) and object-relational features of SQL (from the old Chapter 9).

Then, Chapters 11 and 12 cover XML and systems based on XML. Except for material at the end of the old Chapter 4, which has been moved to Chapter 11, this material is all new. Chapter 11 covers modeling; it includes expanded coverage of DTD's, along with new material on XML Schema. Chapter 12 is devoted to programming, and it includes sections on XPath, XQuery, and XSLT.

Chapter 13 begins the study of database implementation. It covers disk storage and the file structures that are built on disks. This chapter is a condensation of material that, in the first edition, occupied Chapters 11 and 12.

Chapter 14 covers index structures, including B-trees, hashing, and structures for multidimensional indexes. This material also condenses two chapters, 13 and 14, from the first edition.

Chapters 15 and 16 cover query execution and query optimization, respectively. They are similar to the old chapters of the same numbers. Chapter 17 covers logging, and Chapter 18 covers concurrency control; these chapters are also similar to the old chapters with the same numbers. Chapter 19 contains additional topics on concurrency: recovery, deadlocks, and long transactions. This material is a subset of the old Chapter 19.

Chapter 20 is on parallel and distributed databases. In addition to material on parallel query execution from the old Chapter 15 and material on distributed locking and commitment from the old Chapter 19, there are several new sections on distributed query execution: the map-reduce framework for parallel computation, peer-to-peer databases and their implementation of distributed hash tables.

Chapter 21 covers information integration. In addition to material on this subject from the old Chapter 20, we have added a section on local-as-view mediators and a section on entity resolution (finding records from several databases that refer to the same entity, e.g., a person).

Chapter 22 is on data mining. Although there was some material on the subject in the old Chapter 20, almost all of this chapter is new. It covers association rules and frequent itemset mining, including both the famous A-Priori Algorithm and certain efficiency improvements. Chapter 22 includes the key techniques of shingling, minhashing, and locality-sensitive hashing for finding similar items in massive databases, e.g., Web pages that quote substantially

from other Web pages. The chapter concludes with a study of clustering, especially for massive datasets.

Chapter 23, all new, addresses two important ways in which the Internet has impacted database technology. First is search engines, where we discuss algorithms for crawling the Web, the well-known PageRank algorithm for evaluating the importance of Web pages, and its extensions. This chapter also covers data-stream-management systems. We discuss the stream data model and SQL language extensions, and conclude with several interesting algorithms for executing queries on streams.

Prerequisites

We have used the book at the “mezzanine” level, in a sequence of courses taken both by undergraduates and by beginning graduate students. The formal prerequisites for the course are Sophomore-level treatments of:

1. Data structures, algorithms, and discrete math, and
2. Software systems, software engineering, and programming languages.

Of this material, it is important that students have at least a rudimentary understanding of such topics as: algebraic expressions and laws, logic, basic data structures, object-oriented programming concepts, and programming environments. However, we believe that adequate background is acquired by the Junior year of a typical computer science program.

Exercises

The book contains extensive exercises, with some for almost every section. We indicate harder exercises or parts of exercises with an exclamation point. The hardest exercises have a double exclamation point.

Support on the World Wide Web

The book’s home page is

<http://infolab.stanford.edu/~ullman/dscb.html>

You will find errata as we learn of them, and backup materials, including homeworks, projects, and exams. We shall also make available there the sections from the first edition that have been removed from the second.

In addition, there is an accompanying set of on-line homeworks and programming labs using a technology developed by Gradiance Corp. See the section following the Preface for details about the GOAL system. GOAL service

can be purchased at <http://www.prenhall.com/goal>. Instructors who want to use the system in their classes should contact their Prentice-Hall representative or request instructor authorization through the above Web site.

There is a solutions manual for instructors available at

<http://www.prenhall.com/ullman>

This page also gives you access to GOAL and all book materials.

Acknowledgements

We would like to thank Donald Kossmann for helpful discussions, especially concerning XML and its associated programming systems. Also, Bobbie Cochrane assisted us in understanding trigger semantics for a earlier edition.

A large number of people have helped us, either with the development of this book or its predecessors, or by contacting us with errata in the books and/or other Web-based materials. It is our pleasure to acknowledge them all here.

Marc Abromowitz, Joseph H. Adamski, Brad Adelberg, Gleb Ashimov, Donald Aingworth, Teresa Almeida, Brian Babcock, Bruce Baker, Yunfan Bao, Jonathan Becker, Margaret Benitez, Eberhard Bertsch, Larry Bonham, Phillip Bonnet, David Brokaw, Ed Burns, Alex Butler, Karen Butler, Mike Carey, Christopher Chan, Sudarshan Chawathe.

Also Per Christensen, Ed Chang, Surajit Chaudhuri, Ken Chen, Rada Chirkova, Nitin Chopra, Lewis Church, Jr., Bobbie Cochrane, Michael Cole, Alissa Cooper, Arturo Crespo, Linda DeMichiel, Matthew F. Dennis, Tom Dienstbier, Pearl D'Souza, Oliver Duschka, Xavier Faz, Greg Fichtenholtz, Bart Fisher, Simon Frettlöeh, Jarl Friis.

Also John Fry, Chiping Fu, Tracy Fujieda, Prasanna Ganesan, Suzanne Garcia, Mark Gjøl, Manish Godara, Seth Goldberg, Jeff Goldblat, Meredith Goldsmith, Luis Gravano, Gerard Guillemette, Himanshu Gupta, Petri Gynther, Zoltan Gyongyi, Jon Heggland, Rafael Hernandez, Masanori Higashihara, Antti Hjelt, Ben Holtzman, Steve Huntsberry.

Also Sajid Hussain, Leonard Jacobson, Thulasiraman Jeyaraman, Dwight Joe, Brian Jorgensen, Mathew P. Johnson, Sameh Kamel, Jawed Karim, Seth Katz, Pedram Keyani, Victor Kimeli, Ed Knorr, Yeong-Ping Koh, David Koller, Gyorgy Kovacs, Phillip Koza, Brian Kulman, Bill Labiosa, Sang Ho Lee, Young-han Lee, Miguel Licona.

Also Olivier Lobry, Chao-Jun Lu, Waynn Lue, John Manz, Arun Marathe, Philip Minami, Le-Wei Mo, Fabian Modoux, Peter Mork, Mark Mortensen, Ramprakash Narayanaswami, Hankyung Na, Mor Naaman, Mayur Naik, Marie Nilsson, Torbjørn Norbye, Chang-Min Oh, Mehul Patel, Søren Peen, Jian Pei.

Also Xiaobo Peng, Bert Porter, Limbek Reka, Prahash Ramanan, Nisheeth Ranjan, Suzanne Rivoire, Ken Ross, Tim Roughgarden, Mema Roussopoulos, Richard Scherl, Loren Shevitz, Shrikrishna Shrin, June Yoshiko Sison,

PREFACE

ix

Man Cho A. So, Elizabeth Stinson, Qi Su, Ed Swierk, Catherine Tornabene, Anders Uhl, Jonathan Ullman, Mayank Upadhyay.

Also Anatoly Varakin, Vassilis Vassalos, Krishna Venuturimilli, Vikram Vijayaraghavan, Terje Viken, Qiang Wang, Steven Whang, Mike Wiacek, Kristian Widjaja, Janet Wu, Sundar Yamunachari, Takeshi Yokukawa, Bing Yu, Min-Sig Yun, Torben Zahle, Sandy Zhang.

The remaining errors are ours, of course.

H. G.-M.
J. D. U.
J. W.
Stanford, CA
March, 2008

GOAL

Gradiane Online Accelerated Learning (GOAL) is Pearson's premier online homework and assessment system. GOAL is designed to minimize student frustration while providing an interactive teaching experience outside the classroom. (Visit www.prenhall.com/goal for a demonstration and additional information.)

With GOAL's immediate feedback and book-specific hints and pointers, students will have a more efficient and effective learning experience. GOAL delivers immediate assessment and feedback via two kinds of assignments: multiple choice homework exercises and interactive lab projects.

The homework consists of a set of multiple choice questions designed to test student knowledge of a solved problem. When answers are graded as incorrect, students are given a hint and directed back to a specific section in the course textbook for helpful information. Note: Students that are not enrolled in a class may want to enroll in a "Self-Study Course" that allows them to complete the homework exercises on their own.

Unlike syntax checkers and compilers, GOAL's lab projects check for both syntactic and semantic errors. GOAL determines if the student's program runs but more importantly, when checked against a hidden data set, verifies that it returns the correct result. By testing the code and providing immediate feedback, GOAL lets you know exactly which concepts the students have grasped and which ones need to be revisited.

In addition, the GOAL package specific to this book includes programming exercises in SQL and XQuery. Submitted queries are tested for correctness and incorrect results lead to examples of where the query goes wrong. Students can try as many times as they like but writing queries that respond correctly to the examples is not sufficient to get credit for the problem.

Instructors should contact their local Pearson Sales Representative for sales and ordering information for the GOAL Student Access Code and textbook value package.

About the Authors

HECTOR GARCIA-MOLINA is the L. Bosack and S. Lerner Professor of Computer Science and Electrical Engineering at Stanford University. His research interests include digital libraries, information integration, and database application on the Internet. He was a recipient of the SIGMOD Innovations Award and a member of PITAC (President's Information-Technology Advisory Council). He currently serves on the Board of Directors of Oracle Corp.

JEFFREY D. ULLMAN is the Stanford W. Ascherman Professor of Computer Science (emeritus) at Stanford University. He is the author or co-author of 16 books, including *Elements of ML Programming* (Prentice Hall 1998). His research interests include data mining, information integration, and electronic education. He is a member of the National Academy of Engineering, and recipient of a Guggenheim Fellowship, the Karl V. Karlstrom Outstanding Educator Award, the SIGMOD Contributions and Edgar F. Codd Innovations Awards, and the Knuth Prize.

JENNIFER WIDOM is Professor of Computer Science and Electrical Engineering at Stanford University. Her research interests span many aspects of nontraditional data management. She is an ACM Fellow and a member of the National Academy of Engineering, she received the ACM SIGMOD Edgar F. Codd Innovations Award in 2007 and was a Guggenheim Fellow in 2000, and she has served on a variety of program committees, advisory boards, and editorial boards.

Table of Contents

1	The Worlds of Database Systems	1
1.1	The Evolution of Database Systems	1
1.1.1	Early Database Management Systems	2
1.1.2	Relational Database Systems	3
1.1.3	Smaller and Smaller Systems	3
1.1.4	Bigger and Bigger Systems	4
1.1.5	Information Integration	4
1.2	Overview of a Database Management System	5
1.2.1	Data-Definition Language Commands	5
1.2.2	Overview of Query Processing	5
1.2.3	Storage and Buffer Management	7
1.2.4	Transaction Processing	8
1.2.5	The Query Processor	9
1.3	Outline of Database-System Studies	10
1.4	References for Chapter 1	12
I	Relational Database Modeling	15
2	The Relational Model of Data	17
2.1	An Overview of Data Models	17
2.1.1	What is a Data Model?	17
2.1.2	Important Data Models	18
2.1.3	The Relational Model in Brief	18
2.1.4	The Semistructured Model in Brief	19
2.1.5	Other Data Models	20
2.1.6	Comparison of Modeling Approaches	21
2.2	Basics of the Relational Model	21
2.2.1	Attributes	22
2.2.2	Schemas	22
2.2.3	Tuples	22
2.2.4	Domains	23
2.2.5	Equivalent Representations of a Relation	23

2.2.6	Relation Instances	24
2.2.7	Keys of Relations	25
2.2.8	An Example Database Schema	26
2.2.9	Exercises for Section 2.2	28
2.3	Defining a Relation Schema in SQL	29
2.3.1	Relations in SQL	29
2.3.2	Data Types	30
2.3.3	Simple Table Declarations	31
2.3.4	Modifying Relation Schemas	33
2.3.5	Default Values	34
2.3.6	Declaring Keys	34
2.3.7	Exercises for Section 2.3	36
2.4	An Algebraic Query Language	38
2.4.1	Why Do We Need a Special Query Language?	38
2.4.2	What is an Algebra?	38
2.4.3	Overview of Relational Algebra	39
2.4.4	Set Operations on Relations	39
2.4.5	Projection	41
2.4.6	Selection	42
2.4.7	Cartesian Product	43
2.4.8	Natural Joins	43
2.4.9	Theta-Joins	45
2.4.10	Combining Operations to Form Queries	47
2.4.11	Naming and Renaming	49
2.4.12	Relationships Among Operations	50
2.4.13	A Linear Notation for Algebraic Expressions	51
2.4.14	Exercises for Section 2.4	52
2.5	Constraints on Relations	58
2.5.1	Relational Algebra as a Constraint Language	59
2.5.2	Referential Integrity Constraints	59
2.5.3	Key Constraints	60
2.5.4	Additional Constraint Examples	61
2.5.5	Exercises for Section 2.5	62
2.6	Summary of Chapter 2	63
2.7	References for Chapter 2	65
3	Design Theory for Relational Databases	67
3.1	Functional Dependencies	67
3.1.1	Definition of Functional Dependency	68
3.1.2	Keys of Relations	70
3.1.3	Superkeys	71
3.1.4	Exercises for Section 3.1	71
3.2	Rules About Functional Dependencies	72
3.2.1	Reasoning About Functional Dependencies	72
3.2.2	The Splitting/Combining Rule	73

TABLE OF CONTENTS

xv

3.2.3	Trivial Functional Dependencies	74
3.2.4	Computing the Closure of Attributes	75
3.2.5	Why the Closure Algorithm Works	77
3.2.6	The Transitive Rule	79
3.2.7	Closing Sets of Functional Dependencies	80
3.2.8	Projecting Functional Dependencies	81
3.2.9	Exercises for Section 3.2	83
3.3	Design of Relational Database Schemas	85
3.3.1	Anomalies	86
3.3.2	Decomposing Relations	86
3.3.3	Boyce-Codd Normal Form	88
3.3.4	Decomposition into BCNF	89
3.3.5	Exercises for Section 3.3	92
3.4	Decomposition: The Good, Bad, and Ugly	93
3.4.1	Recovering Information from a Decomposition	94
3.4.2	The Chase Test for Lossless Join	96
3.4.3	Why the Chase Works	99
3.4.4	Dependency Preservation	100
3.4.5	Exercises for Section 3.4	102
3.5	Third Normal Form	102
3.5.1	Definition of Third Normal Form	102
3.5.2	The Synthesis Algorithm for 3NF Schemas	103
3.5.3	Why the 3NF Synthesis Algorithm Works	104
3.5.4	Exercises for Section 3.5	105
3.6	Multivalued Dependencies	105
3.6.1	Attribute Independence and Its Consequent Redundancy	106
3.6.2	Definition of Multivalued Dependencies	107
3.6.3	Reasoning About Multivalued Dependencies	108
3.6.4	Fourth Normal Form	110
3.6.5	Decomposition into Fourth Normal Form	111
3.6.6	Relationships Among Normal Forms	113
3.6.7	Exercises for Section 3.6	113
3.7	An Algorithm for Discovering MVD's	115
3.7.1	The Closure and the Chase	115
3.7.2	Extending the Chase to MVD's	116
3.7.3	Why the Chase Works for MVD's	118
3.7.4	Projecting MVD's	119
3.7.5	Exercises for Section 3.7	120
3.8	Summary of Chapter 3	121
3.9	References for Chapter 3	122

4	High-Level Database Models	125
4.1	The Entity/Relationship Model	126
4.1.1	Entity Sets	126
4.1.2	Attributes	126
4.1.3	Relationships	127
4.1.4	Entity-Relationship Diagrams	127
4.1.5	Instances of an E/R Diagram	128
4.1.6	Multiplicity of Binary E/R Relationships	129
4.1.7	Multiway Relationships	130
4.1.8	Roles in Relationships	131
4.1.9	Attributes on Relationships	134
4.1.10	Converting Multiway Relationships to Binary	134
4.1.11	Subclasses in the E/R Model	135
4.1.12	Exercises for Section 4.1	138
4.2	Design Principles	140
4.2.1	Faithfulness	140
4.2.2	Avoiding Redundancy	141
4.2.3	Simplicity Counts	142
4.2.4	Choosing the Right Relationships	142
4.2.5	Picking the Right Kind of Element	144
4.2.6	Exercises for Section 4.2	145
4.3	Constraints in the E/R Model	148
4.3.1	Keys in the E/R Model	148
4.3.2	Representing Keys in the E/R Model	149
4.3.3	Referential Integrity	150
4.3.4	Degree Constraints	151
4.3.5	Exercises for Section 4.3	151
4.4	Weak Entity Sets	152
4.4.1	Causes of Weak Entity Sets	152
4.4.2	Requirements for Weak Entity Sets	153
4.4.3	Weak Entity Set Notation	155
4.4.4	Exercises for Section 4.4	156
4.5	From E/R Diagrams to Relational Designs	157
4.5.1	From Entity Sets to Relations	157
4.5.2	From E/R Relationships to Relations	158
4.5.3	Combining Relations	160
4.5.4	Handling Weak Entity Sets	161
4.5.5	Exercises for Section 4.5	163
4.6	Converting Subclass Structures to Relations	165
4.6.1	E/R-Style Conversion	166
4.6.2	An Object-Oriented Approach	167
4.6.3	Using Null Values to Combine Relations	168
4.6.4	Comparison of Approaches	169
4.6.5	Exercises for Section 4.6	171
4.7	Unified Modeling Language	171

4.7.1	UML Classes	172
4.7.2	Keys for UML classes	173
4.7.3	Associations	173
4.7.4	Self-Associations	175
4.7.5	Association Classes	175
4.7.6	Subclasses in UML	176
4.7.7	Aggregations and Compositions	177
4.7.8	Exercises for Section 4.7	179
4.8	From UML Diagrams to Relations	179
4.8.1	UML-to-Relations Basics	179
4.8.2	From UML Subclasses to Relations	180
4.8.3	From Aggregations and Compositions to Relations	181
4.8.4	The UML Analog of Weak Entity Sets	181
4.8.5	Exercises for Section 4.8	183
4.9	Object Definition Language	183
4.9.1	Class Declarations	184
4.9.2	Attributes in ODL	184
4.9.3	Relationships in ODL	185
4.9.4	Inverse Relationships	186
4.9.5	Multiplicity of Relationships	186
4.9.6	Types in ODL	188
4.9.7	Subclasses in ODL	190
4.9.8	Declaring Keys in ODL	191
4.9.9	Exercises for Section 4.9	192
4.10	From ODL Designs to Relational Designs	193
4.10.1	From ODL Classes to Relations	193
4.10.2	Complex Attributes in Classes	194
4.10.3	Representing Set-Valued Attributes	195
4.10.4	Representing Other Type Constructors	196
4.10.5	Representing ODL Relationships	198
4.10.6	Exercises for Section 4.10	198
4.11	Summary of Chapter 4	200
4.12	References for Chapter 4	202

II. Relational Database Programming 203

5 Algebraic and Logical Query Languages 205

5.1	Relational Operations on Bags	205
5.1.1	Why Bags?	206
5.1.2	Union, Intersection, and Difference of Bags	207
5.1.3	Projection of Bags	208
5.1.4	Selection on Bags	209
5.1.5	Product of Bags	210
5.1.6	Joins of Bags	210

5.1.7	Exercises for Section 5.1	212
5.2	Extended Operators of Relational Algebra	213
5.2.1	Duplicate Elimination	214
5.2.2	Aggregation Operators	214
5.2.3	Grouping	215
5.2.4	The Grouping Operator	216
5.2.5	Extending the Projection Operator	217
5.2.6	The Sorting Operator	219
5.2.7	Outerjoins	219
5.2.8	Exercises for Section 5.2	222
5.3	A Logic for Relations	222
5.3.1	Predicates and Atoms	223
5.3.2	Arithmetic Atoms	223
5.3.3	Datalog Rules and Queries	224
5.3.4	Meaning of Datalog Rules	225
5.3.5	Extensional and Intensional Predicates	228
5.3.6	Datalog Rules Applied to Bags	228
5.3.7	Exercises for Section 5.3	230
5.4	Relational Algebra and Datalog	230
5.4.1	Boolean Operations	231
5.4.2	Projection	232
5.4.3	Selection	232
5.4.4	Product	235
5.4.5	Joins	235
5.4.6	Simulating Multiple Operations with Datalog	236
5.4.7	Comparison Between Datalog and Relational Algebra	238
5.4.8	Exercises for Section 5.4	238
5.5	Summary of Chapter 5	240
5.6	References for Chapter 5	241
6	The Database Language SQL	243
6.1	Simple Queries in SQL	244
6.1.1	Projection in SQL	246
6.1.2	Selection in SQL	248
6.1.3	Comparison of Strings	250
6.1.4	Pattern Matching in SQL	250
6.1.5	Dates and Times	251
6.1.6	Null Values and Comparisons Involving NULL	252
6.1.7	The Truth-Value UNKNOWN	253
6.1.8	Ordering the Output	255
6.1.9	Exercises for Section 6.1	256
6.2	Queries Involving More Than One Relation	258
6.2.1	Products and Joins in SQL	259
6.2.2	Disambiguating Attributes	260
6.2.3	Tuple Variables	261

6.2.4	Interpreting Multirelation Queries	262
6.2.5	Union, Intersection, and Difference of Queries	265
6.2.6	Exercises for Section 6.2	267
6.3	Subqueries	268
6.3.1	Subqueries that Produce Scalar Values	269
6.3.2	Conditions Involving Relations	270
6.3.3	Conditions Involving Tuples	271
6.3.4	Correlated Subqueries	273
6.3.5	Subqueries in FROM Clauses	274
6.3.6	SQL Join Expressions	275
6.3.7	Natural Joins	276
6.3.8	Outerjoins	277
6.3.9	Exercises for Section 6.3	279
6.4	Full-Relation Operations	281
6.4.1	Eliminating Duplicates	281
6.4.2	Duplicates in Unions, Intersections, and Differences	282
6.4.3	Grouping and Aggregation in SQL	283
6.4.4	Aggregation Operators	284
6.4.5	Grouping	285
6.4.6	Grouping, Aggregation, and Nulls	287
6.4.7	HAVING Clauses	288
6.4.8	Exercises for Section 6.4	289
6.5	Database Modifications	291
6.5.1	Insertion	291
6.5.2	Deletion	292
6.5.3	Updates	294
6.5.4	Exercises for Section 6.5	295
6.6	Transactions in SQL	296
6.6.1	Serializability	296
6.6.2	Atomicity	298
6.6.3	Transactions	299
6.6.4	Read-Only Transactions	300
6.6.5	Dirty Reads	302
6.6.6	Other Isolation Levels	304
6.6.7	Exercises for Section 6.6	306
6.7	Summary of Chapter 6	307
6.8	References for Chapter 6	308
7	Constraints and Triggers	311
7.1	Keys and Foreign Keys	311
7.1.1	Declaring Foreign-Key Constraints	312
7.1.2	Maintaining Referential Integrity	313
7.1.3	Deferred Checking of Constraints	315
7.1.4	Exercises for Section 7.1	318
7.2	Constraints on Attributes and Tuples	319

7.2.1	Not-Null Constraints	319
7.2.2	Attribute-Based CHECK Constraints	320
7.2.3	Tuple-Based CHECK Constraints	321
7.2.4	Comparison of Tuple- and Attribute-Based Constraints	323
7.2.5	Exercises for Section 7.2	323
7.3	Modification of Constraints	325
7.3.1	Giving Names to Constraints	325
7.3.2	Altering Constraints on Tables	326
7.3.3	Exercises for Section 7.3	327
7.4	Assertions	328
7.4.1	Creating Assertions	328
7.4.2	Using Assertions	329
7.4.3	Exercises for Section 7.4	330
7.5	Triggers	332
7.5.1	Triggers in SQL	332
7.5.2	The Options for Trigger Design	334
7.5.3	Exercises for Section 7.5	337
7.6	Summary of Chapter 7	339
7.7	References for Chapter 7	339
8	Views and Indexes	341
8.1	Virtual Views	341
8.1.1	Declaring Views	341
8.1.2	Querying Views	343
8.1.3	Renaming Attributes	343
8.1.4	Exercises for Section 8.1	344
8.2	Modifying Views	344
8.2.1	View Removal	345
8.2.2	Updatable Views	345
8.2.3	Instead-Of Triggers on Views	347
8.2.4	Exercises for Section 8.2	349
8.3	Indexes in SQL	350
8.3.1	Motivation for Indexes	350
8.3.2	Declaring Indexes	351
8.3.3	Exercises for Section 8.3	352
8.4	Selection of Indexes	352
8.4.1	A Simple Cost Model	352
8.4.2	Some Useful Indexes	353
8.4.3	Calculating the Best Indexes to Create	355
8.4.4	Automatic Selection of Indexes to Create	357
8.4.5	Exercises for Section 8.4	359
8.5	Materialized Views	359
8.5.1	Maintaining a Materialized View	360
8.5.2	Periodic Maintenance of Materialized Views	362
8.5.3	Rewriting Queries to Use Materialized Views	362

8.5.4	Automatic Creation of Materialized Views	364
8.5.5	Exercises for Section 8.5	365
8.6	Summary of Chapter 8	366
8.7	References for Chapter 8	367
9	SQL in a Server Environment	369
9.1	The Three-Tier Architecture	369
9.1.1	The Web-Server Tier	370
9.1.2	The Application Tier	371
9.1.3	The Database Tier	372
9.2	The SQL Environment	372
9.2.1	Environments	373
9.2.2	Schemas	374
9.2.3	Catalogs	375
9.2.4	Clients and Servers in the SQL Environment	375
9.2.5	Connections	376
9.2.6	Sessions	377
9.2.7	Modules	378
9.3	The SQL/Host-Language Interface	378
9.3.1	The Impedance Mismatch Problem	380
9.3.2	Connecting SQL to the Host Language	380
9.3.3	The DECLARE Section	381
9.3.4	Using Shared Variables	382
9.3.5	Single-Row Select Statements	383
9.3.6	Cursors	383
9.3.7	Modifications by Cursor	386
9.3.8	Protecting Against Concurrent Updates	387
9.3.9	Dynamic SQL	388
9.3.10	Exercises for Section 9.3	390
9.4	Stored Procedures	391
9.4.1	Creating PSM Functions and Procedures	391
9.4.2	Some Simple Statement Forms in PSM	392
9.4.3	Branching Statements	394
9.4.4	Queries in PSM	395
9.4.5	Loops in PSM	396
9.4.6	For-Loops	398
9.4.7	Exceptions in PSM	400
9.4.8	Using PSM Functions and Procedures	402
9.4.9	Exercises for Section 9.4	402
9.5	Using a Call-Level Interface	404
9.5.1	Introduction to SQL/CLI	405
9.5.2	Processing Statements	407
9.5.3	Fetching Data From a Query Result	408
9.5.4	Passing Parameters to Queries	410
9.5.5	Exercises for Section 9.5	412

9.6	JDBC	412
9.6.1	Introduction to JDBC	412
9.6.2	Creating Statements in JDBC	413
9.6.3	Cursor Operations in JDBC	415
9.6.4	Parameter Passing	416
9.6.5	Exercises for Section 9.6	416
9.7	PHP	416
9.7.1	PHP Basics	417
9.7.2	Arrays	418
9.7.3	The PEAR DB Library	419
9.7.4	Creating a Database Connection Using DB	419
9.7.5	Executing SQL Statements	419
9.7.6	Cursor Operations in PHP	420
9.7.7	Dynamic SQL in PHP	421
9.7.8	Exercises for Section 9.7	422
9.8	Summary of Chapter 9	422
9.9	References for Chapter 9	423
10	Advanced Topics in Relational Databases	425
10.1	Security and User Authorization in SQL	425
10.1.1	Privileges	426
10.1.2	Creating Privileges	427
10.1.3	The Privilege-Checking Process	428
10.1.4	Granting Privileges	430
10.1.5	Grant Diagrams	431
10.1.6	Revoking Privileges	433
10.1.7	Exercises for Section 10.1	436
10.2	Recursion in SQL	437
10.2.1	Defining Recursive Relations in SQL	437
10.2.2	Problematic Expressions in Recursive SQL	440
10.2.3	Exercises for Section 10.2	443
10.3	The Object-Relational Model	445
10.3.1	From Relations to Object-Relations	445
10.3.2	Nested Relations	446
10.3.3	References	447
10.3.4	Object-Oriented Versus Object-Relational	449
10.3.5	Exercises for Section 10.3	450
10.4	User-Defined Types in SQL	451
10.4.1	Defining Types in SQL	451
10.4.2	Method Declarations in UDT's	452
10.4.3	Method Definitions	453
10.4.4	Declaring Relations with a UDT	454
10.4.5	References	454
10.4.6	Creating Object ID's for Tables	455
10.4.7	Exercises for Section 10.4	457

10.5 Operations on Object-Relational Data	457
10.5.1 Following References	457
10.5.2 Accessing Components of Tuples with a UDT	458
10.5.3 Generator and Mutator Functions	460
10.5.4 Ordering Relationships on UDT's	461
10.5.5 Exercises for Section 10.5	463
10.6 On-Line Analytic Processing	464
10.6.1 OLAP and Data Warehouses	465
10.6.2 OLAP Applications	465
10.6.3 A Multidimensional View of OLAP Data	466
10.6.4 Star Schemas	467
10.6.5 Slicing and Dicing	469
10.6.6 Exercises for Section 10.6	472
10.7 Data Cubes	473
10.7.1 The Cube Operator	473
10.7.2 The Cube Operator in SQL	475
10.7.3 Exercises for Section 10.7	477
10.8 Summary of Chapter 10	478
10.9 References for Chapter 10	480
 III Modeling and Programming for Semistructured Data	 481
11 The Semistructured-Data Model	483
11.1 Semistructured Data	483
11.1.1 Motivation for the Semistructured-Data Model	483
11.1.2 Semistructured Data Representation	484
11.1.3 Information Integration Via Semistructured Data	486
11.1.4 Exercises for Section 11.1	487
11.2 XML	488
11.2.1 Semantic Tags	488
11.2.2 XML With and Without a Schema	489
11.2.3 Well-Formed XML	489
11.2.4 Attributes	490
11.2.5 Attributes That Connect Elements	491
11.2.6 Namespaces	493
11.2.7 XML and Databases	493
11.2.8 Exercises for Section 11.2	495
11.3 Document Type Definitions	495
11.3.1 The Form of a DTD	495
11.3.2 Using a DTD	499
11.3.3 Attribute Lists	499
11.3.4 Identifiers and References	500
11.3.5 Exercises for Section 11.3	502

11.4 XML Schema	502
11.4.1 The Form of an XML Schema	502
11.4.2 Elements	503
11.4.3 Complex Types	504
11.4.4 Attributes	506
11.4.5 Restricted Simple Types	507
11.4.6 Keys in XML Schema	509
11.4.7 Foreign Keys in XML Schema	510
11.4.8 Exercises for Section 11.4	512
11.5 Summary of Chapter 11	514
11.6 References for Chapter 11	515
12 Programming Languages for XML	517
12.1 XPath	517
12.1.1 The XPath Data Model	518
12.1.2 Document Nodes	519
12.1.3 Path Expressions	519
12.1.4 Relative Path Expressions	521
12.1.5 Attributes in Path Expressions	521
12.1.6 Axes	521
12.1.7 Context of Expressions	522
12.1.8 Wildcards	523
12.1.9 Conditions in Path Expressions	523
12.1.10 Exercises for Section 12.1	526
12.2 XQuery	528
12.2.1 XQuery Basics	530
12.2.2 FLWR Expressions	530
12.2.3 Replacement of Variables by Their Values	534
12.2.4 Joins in XQuery	536
12.2.5 XQuery Comparison Operators	537
12.2.6 Elimination of Duplicates	538
12.2.7 Quantification in XQuery	539
12.2.8 Aggregations	540
12.2.9 Branching in XQuery Expressions	540
12.2.10 Ordering the Result of a Query	541
12.2.11 Exercises for Section 12.2	543
12.3 Extensible Stylesheet Language	544
12.3.1 XSLT Basics	544
12.3.2 Templates	544
12.3.3 Obtaining Values From XML Data	545
12.3.4 Recursive Use of Templates	546
12.3.5 Iteration in XSLT	549
12.3.6 Conditionals in XSLT	551
12.3.7 Exercises for Section 12.3	551
12.4 Summary of Chapter 12	553

12.5 References for Chapter 12	554
--	-----

IV Database System Implementation 555

13 Secondary Storage Management	557
13.1 The Memory Hierarchy	557
13.1.1 The Memory Hierarchy	557
13.1.2 Transfer of Data Between Levels	560
13.1.3 Volatile and Nonvolatile Storage	560
13.1.4 Virtual Memory	560
13.1.5 Exercises for Section 13.1	561
13.2 Disks	562
13.2.1 Mechanics of Disks	562
13.2.2 The Disk Controller	564
13.2.3 Disk Access Characteristics	564
13.2.4 Exercises for Section 13.2	567
13.3 Accelerating Access to Secondary Storage	568
13.3.1 The I/O Model of Computation	568
13.3.2 Organizing Data by Cylinders	569
13.3.3 Using Multiple Disks	570
13.3.4 Mirroring Disks	571
13.3.5 Disk Scheduling and the Elevator Algorithm	571
13.3.6 Prefetching and Large-Scale Buffering	573
13.3.7 Exercises for Section 13.3	573
13.4 Disk Failures	575
13.4.1 Intermittent Failures	576
13.4.2 Checksums	576
13.4.3 Stable Storage	577
13.4.4 Error-Handling Capabilities of Stable Storage	578
13.4.5 Recovery from Disk Crashes	578
13.4.6 Mirroring as a Redundancy Technique	579
13.4.7 Parity Blocks	580
13.4.8 An Improvement: RAID 5	583
13.4.9 Coping With Multiple Disk Crashes	584
13.4.10 Exercises for Section 13.4	587
13.5 Arranging Data on Disk	590
13.5.1 Fixed-Length Records	590
13.5.2 Packing Fixed-Length Records into Blocks	592
13.5.3 Exercises for Section 13.5	593
13.6 Representing Block and Record Addresses	593
13.6.1 Addresses in Client-Server Systems	593
13.6.2 Logical and Structured Addresses	595
13.6.3 Pointer Swizzling	596
13.6.4 Returning Blocks to Disk	600

13.6.5 Pinned Records and Blocks	600
13.6.6 Exercises for Section 13.6	602
13.7 Variable-Length Data and Records	603
13.7.1 Records With Variable-Length Fields	604
13.7.2 Records With Repeating Fields	605
13.7.3 Variable-Format Records	607
13.7.4 Records That Do Not Fit in a Block	608
13.7.5 BLOBs	608
13.7.6 Column Stores	609
13.7.7 Exercises for Section 13.7	610
13.8 Record Modifications	612
13.8.1 Insertion	612
13.8.2 Deletion	614
13.8.3 Update	615
13.8.4 Exercises for Section 13.8	615
13.9 Summary of Chapter 13	615
13.10 References for Chapter 13	617
14 Index Structures	619
14.1 Index-Structure Basics	620
14.1.1 Sequential Files	621
14.1.2 Dense Indexes	621
14.1.3 Sparse Indexes	622
14.1.4 Multiple Levels of Index	623
14.1.5 Secondary Indexes	624
14.1.6 Applications of Secondary Indexes	625
14.1.7 Indirection in Secondary Indexes	626
14.1.8 Document Retrieval and Inverted Indexes	628
14.1.9 Exercises for Section 14.1	631
14.2 B-Trees	633
14.2.1 The Structure of B-trees	634
14.2.2 Applications of B-trees	637
14.2.3 Lookup in B-Trees	639
14.2.4 Range Queries	639
14.2.5 Insertion Into B-Trees	640
14.2.6 Deletion From B-Trees	642
14.2.7 Efficiency of B-Trees	645
14.2.8 Exercises for Section 14.2	646
14.3 Hash Tables	648
14.3.1 Secondary-Storage Hash Tables	649
14.3.2 Insertion Into a Hash Table	649
14.3.3 Hash-Table Deletion	650
14.3.4 Efficiency of Hash Table Indexes	651
14.3.5 Extensible Hash Tables	652
14.3.6 Insertion Into Extensible Hash Tables	653

14.3.7	Linear Hash Tables	655
14.3.8	Insertion Into Linear Hash Tables	657
14.3.9	Exercises for Section 14.3	659
14.4	Multidimensional Indexes	661
14.4.1	Applications of Multidimensional Indexes	661
14.4.2	Executing Range Queries Using Conventional Indexes	663
14.4.3	Executing Nearest-Neighbor Queries Using Conventional Indexes	664
14.4.4	Overview of Multidimensional Index Structures	664
14.5	Hash Structures for Multidimensional Data	665
14.5.1	Grid Files	665
14.5.2	Lookup in a Grid File	666
14.5.3	Insertion Into Grid Files	667
14.5.4	Performance of Grid Files	669
14.5.5	Partitioned Hash Functions	671
14.5.6	Comparison of Grid Files and Partitioned Hashing	673
14.5.7	Exercises for Section 14.5	673
14.6	Tree Structures for Multidimensional Data	675
14.6.1	Multiple-Key Indexes	675
14.6.2	Performance of Multiple-Key Indexes	676
14.6.3	<i>kd</i> -Trees	677
14.6.4	Operations on <i>kd</i> -Trees	679
14.6.5	Adapting <i>kd</i> -Trees to Secondary Storage	681
14.6.6	Quad Trees	681
14.6.7	R-Trees	683
14.6.8	Operations on R-Trees	684
14.6.9	Exercises for Section 14.6	686
14.7	Bitmap Indexes	688
14.7.1	Motivation for Bitmap Indexes	689
14.7.2	Compressed Bitmaps	691
14.7.3	Operating on Run-Length-Encoded Bit-Vectors	693
14.7.4	Managing Bitmap Indexes	693
14.7.5	Exercises for Section 14.7	695
14.8	Summary of Chapter 14	695
14.9	References for Chapter 14	697
15	Query Execution	701
15.1	Introduction to Physical-Query-Plan Operators	703
15.1.1	Scanning Tables	703
15.1.2	Sorting While Scanning Tables	704
15.1.3	The Computation Model for Physical Operators	704
15.1.4	Parameters for Measuring Costs	705
15.1.5	I/O Cost for Scan Operators	706
15.1.6	Iterators for Implementation of Physical Operators	707
15.2	One-Pass Algorithms	709

15.2.1	One-Pass Algorithms for Tuple-at-a-Time Operations . .	711
15.2.2	One-Pass Algorithms for Unary, Full-Relation Operations	712
15.2.3	One-Pass Algorithms for Binary Operations	715
15.2.4	Exercises for Section 15.2	718
15.3	Nested-Loop Joins	718
15.3.1	Tuple-Based Nested-Loop Join	719
15.3.2	An Iterator for Tuple-Based Nested-Loop Join	719
15.3.3	Block-Based Nested-Loop Join Algorithm	719
15.3.4	Analysis of Nested-Loop Join	721
15.3.5	Summary of Algorithms so Far	722
15.3.6	Exercises for Section 15.3	722
15.4	Two-Pass Algorithms Based on Sorting	723
15.4.1	Two-Phase, Multiway Merge-Sort	723
15.4.2	Duplicate Elimination Using Sorting	725
15.4.3	Grouping and Aggregation Using Sorting	726
15.4.4	A Sort-Based Union Algorithm	726
15.4.5	Sort-Based Intersection and Difference	727
15.4.6	A Simple Sort-Based Join Algorithm	728
15.4.7	Analysis of Simple Sort-Join	729
15.4.8	A More Efficient Sort-Based Join	729
15.4.9	Summary of Sort-Based Algorithms	730
15.4.10	Exercises for Section 15.4	730
15.5	Two-Pass Algorithms Based on Hashing	732
15.5.1	Partitioning Relations by Hashing	732
15.5.2	A Hash-Based Algorithm for Duplicate Elimination	732
15.5.3	Hash-Based Grouping and Aggregation	733
15.5.4	Hash-Based Union, Intersection, and Difference	734
15.5.5	The Hash-Join Algorithm	734
15.5.6	Saving Some Disk I/O's	735
15.5.7	Summary of Hash-Based Algorithms	737
15.5.8	Exercises for Section 15.5	738
15.6	Index-Based Algorithms	739
15.6.1	Clustering and Nonclustering Indexes	739
15.6.2	Index-Based Selection	740
15.6.3	Joining by Using an Index	742
15.6.4	Joins Using a Sorted Index	743
15.6.5	Exercises for Section 15.6	745
15.7	Buffer Management	746
15.7.1	Buffer Management Architecture	746
15.7.2	Buffer Management Strategies	747
15.7.3	The Relationship Between Physical Operator Selection and Buffer Management	750
15.7.4	Exercises for Section 15.7	751
15.8	Algorithms Using More Than Two Passes	752
15.8.1	Multipass Sort-Based Algorithms	752

15.8.2	Performance of Multipass, Sort-Based Algorithms	753
15.8.3	Multipass Hash-Based Algorithms	754
15.8.4	Performance of Multipass Hash-Based Algorithms	754
15.8.5	Exercises for Section 15.8	755
15.9	Summary of Chapter 15	756
15.10	References for Chapter 15	757
16	The Query Compiler	759
16.1	Parsing and Preprocessing	760
16.1.1	Syntax Analysis and Parse Trees	760
16.1.2	A Grammar for a Simple Subset of SQL	761
16.1.3	The Preprocessor	764
16.1.4	Preprocessing Queries Involving Views	765
16.1.5	Exercises for Section 16.1	767
16.2	Algebraic Laws for Improving Query Plans	768
16.2.1	Commutative and Associative Laws	768
16.2.2	Laws Involving Selection	770
16.2.3	Pushing Selections	772
16.2.4	Laws Involving Projection	774
16.2.5	Laws About Joins and Products	776
16.2.6	Laws Involving Duplicate Elimination	777
16.2.7	Laws Involving Grouping and Aggregation	777
16.2.8	Exercises for Section 16.2	780
16.3	From Parse Trees to Logical Query Plans	781
16.3.1	Conversion to Relational Algebra	782
16.3.2	Removing Subqueries From Conditions	783
16.3.3	Improving the Logical Query Plan	788
16.3.4	Grouping Associative/Commutative Operators	790
16.3.5	Exercises for Section 16.3	791
16.4	Estimating the Cost of Operations	792
16.4.1	Estimating Sizes of Intermediate Relations	793
16.4.2	Estimating the Size of a Projection	794
16.4.3	Estimating the Size of a Selection	794
16.4.4	Estimating the Size of a Join	797
16.4.5	Natural Joins With Multiple Join Attributes	799
16.4.6	Joins of Many Relations	800
16.4.7	Estimating Sizes for Other Operations	801
16.4.8	Exercises for Section 16.4	802
16.5	Introduction to Cost-Based Plan Selection	803
16.5.1	Obtaining Estimates for Size Parameters	804
16.5.2	Computation of Statistics	807
16.5.3	Heuristics for Reducing the Cost of Logical Query Plans	808
16.5.4	Approaches to Enumerating Physical Plans	810
16.5.5	Exercises for Section 16.5	813
16.6	Choosing an Order for Joins	814

16.6.1	Significance of Left and Right Join Arguments	815
16.6.2	Join Trees	815
16.6.3	Left-Deep Join Trees	816
16.6.4	Dynamic Programming to Select a Join Order and Grouping	819
16.6.5	Dynamic Programming With More Detailed Cost Functions	823
16.6.6	A Greedy Algorithm for Selecting a Join Order	824
16.6.7	Exercises for Section 16.6	825
16.7	Completing the Physical-Query-Plan	826
16.7.1	Choosing a Selection Method	827
16.7.2	Choosing a Join Method	829
16.7.3	Pipelining Versus Materialization	830
16.7.4	Pipelining Unary Operations	830
16.7.5	Pipelining Binary Operations	830
16.7.6	Notation for Physical Query Plans	834
16.7.7	Ordering of Physical Operations	837
16.7.8	Exercises for Section 16.7	838
16.8	Summary of Chapter 16	839
16.9	References for Chapter 16	841
17	Coping With System Failures	843
17.1	Issues and Models for Resilient Operation	843
17.1.1	Failure Modes	844
17.1.2	More About Transactions	845
17.1.3	Correct Execution of Transactions	846
17.1.4	The Primitive Operations of Transactions	848
17.1.5	Exercises for Section 17.1	851
17.2	Undo Logging	851
17.2.1	Log Records	851
17.2.2	The Undo-Logging Rules	853
17.2.3	Recovery Using Undo Logging	855
17.2.4	Checkpointing	857
17.2.5	Nonquiescent Checkpointing	858
17.2.6	Exercises for Section 17.2	862
17.3	Redo Logging	863
17.3.1	The Redo-Logging Rule	863
17.3.2	Recovery With Redo Logging	864
17.3.3	Checkpointing a Redo Log	866
17.3.4	Recovery With a Checkpointed Redo Log	867
17.3.5	Exercises for Section 17.3	868
17.4	Undo/Redo Logging	869
17.4.1	The Undo/Redo Rules	870
17.4.2	Recovery With Undo/Redo Logging	870
17.4.3	Checkpointing an Undo/Redo Log	872
17.4.4	Exercises for Section 17.4	874
17.5	Protecting Against Media Failures	875

TABLE OF CONTENTS

xxxi

17.5.1	The Archive	875
17.5.2	Nonquiescent Archiving	875
17.5.3	Recovery Using an Archive and Log	878
17.5.4	Exercises for Section 17.5	879
17.6	Summary of Chapter 17	879
17.7	References for Chapter 17	881
18	Concurrency Control	883
18.1	Serial and Serializable Schedules	884
18.1.1	Schedules	884
18.1.2	Serial Schedules	885
18.1.3	Serializable Schedules	886
18.1.4	The Effect of Transaction Semantics	887
18.1.5	A Notation for Transactions and Schedules	889
18.1.6	Exercises for Section 18.1	889
18.2	Conflict-Serializability	890
18.2.1	Conflicts	890
18.2.2	Precedence Graphs and a Test for Conflict-Serializability	892
18.2.3	Why the Precedence-Graph Test Works	894
18.2.4	Exercises for Section 18.2	895
18.3	Enforcing Serializability by Locks	897
18.3.1	Locks	898
18.3.2	The Locking Scheduler	900
18.3.3	Two-Phase Locking	900
18.3.4	Why Two-Phase Locking Works	901
18.3.5	Exercises for Section 18.3	903
18.4	Locking Systems With Several Lock Modes	905
18.4.1	Shared and Exclusive Locks	905
18.4.2	Compatibility Matrices	907
18.4.3	Upgrading Locks	908
18.4.4	Update Locks	909
18.4.5	Increment Locks	911
18.4.6	Exercises for Section 18.4	913
18.5	An Architecture for a Locking Scheduler	915
18.5.1	A Scheduler That Inserts Lock Actions	915
18.5.2	The Lock Table	918
18.5.3	Exercises for Section 18.5	921
18.6	Hierarchies of Database Elements	921
18.6.1	Locks With Multiple Granularity	921
18.6.2	Warning Locks	922
18.6.3	Phantoms and Handling Insertions Correctly	926
18.6.4	Exercises for Section 18.6	927
18.7	The Tree Protocol	927
18.7.1	Motivation for Tree-Based Locking	927
18.7.2	Rules for Access to Tree-Structured Data	928

18.7.3	Why the Tree Protocol Works	929
18.7.4	Exercises for Section 18.7	932
18.8	Concurrency Control by Timestamps	933
18.8.1	Timestamps	934
18.8.2	Physically Unrealizable Behaviors	934
18.8.3	Problems With Dirty Data	935
18.8.4	The Rules for Timestamp-Based Scheduling	937
18.8.5	Multiversion Timestamps	939
18.8.6	Timestamps Versus Locking	941
18.8.7	Exercises for Section 18.8	942
18.9	Concurrency Control by Validation	942
18.9.1	Architecture of a Validation-Based Scheduler	942
18.9.2	The Validation Rules	943
18.9.3	Comparison of Three Concurrency-Control Mechanisms	946
18.9.4	Exercises for Section 18.9	948
18.10	Summary of Chapter 18	948
18.11	References for Chapter 18	950
19	More About Transaction Management	953
19.1	Serializability and Recoverability	953
19.1.1	The Dirty-Data Problem	954
19.1.2	Cascading Rollback	955
19.1.3	Recoverable Schedules	956
19.1.4	Schedules That Avoid Cascading Rollback	957
19.1.5	Managing Rollbacks Using Locking	957
19.1.6	Group Commit	959
19.1.7	Logical Logging	960
19.1.8	Recovery From Logical Logs	963
19.1.9	Exercises for Section 19.1	965
19.2	Deadlocks	966
19.2.1	Deadlock Detection by Timeout	967
19.2.2	The Waits-For Graph	967
19.2.3	Deadlock Prevention by Ordering Elements	970
19.2.4	Detecting Deadlocks by Timestamps	970
19.2.5	Comparison of Deadlock-Management Methods	972
19.2.6	Exercises for Section 19.2	974
19.3	Long-Duration Transactions	975
19.3.1	Problems of Long Transactions	976
19.3.2	Sagas	978
19.3.3	Compensating Transactions	979
19.3.4	Why Compensating Transactions Work	980
19.3.5	Exercises for Section 19.3	981
19.4	Summary of Chapter 19	982
19.5	References for Chapter 19	983

20 Parallel and Distributed Databases	985
20.1 Parallel Algorithms on Relations	985
20.1.1 Models of Parallelism	986
20.1.2 Tuple-at-a-Time Operations in Parallel	989
20.1.3 Parallel Algorithms for Full-Relation Operations	989
20.1.4 Performance of Parallel Algorithms	990
20.1.5 Exercises for Section 20.1	993
20.2 The Map-Reduce Parallelism Framework	993
20.2.1 The Storage Model	993
20.2.2 The Map Function	994
20.2.3 The Reduce Function	995
20.2.4 Exercises for Section 20.2	996
20.3 Distributed Databases	997
20.3.1 Distribution of Data	997
20.3.2 Distributed Transactions	998
20.3.3 Data Replication	999
20.3.4 Exercises for Section 20.3	1000
20.4 Distributed Query Processing	1000
20.4.1 The Distributed Join Problem	1000
20.4.2 Semijoin Reductions	1001
20.4.3 Joins of Many Relations	1002
20.4.4 Acyclic Hypergraphs	1003
20.4.5 Full Reducers for Acyclic Hypergraphs	1005
20.4.6 Why the Full-Reducer Algorithm Works	1006
20.4.7 Exercises for Section 20.4	1007
20.5 Distributed Commit	1008
20.5.1 Supporting Distributed Atomicity	1008
20.5.2 Two-Phase Commit	1009
20.5.3 Recovery of Distributed Transactions	1011
20.5.4 Exercises for Section 20.5	1013
20.6 Distributed Locking	1014
20.6.1 Centralized Lock Systems	1015
20.6.2 A Cost Model for Distributed Locking Algorithms	1015
20.6.3 Locking Replicated Elements	1016
20.6.4 Primary-Copy Locking	1017
20.6.5 Global Locks From Local Locks	1017
20.6.6 Exercises for Section 20.6	1019
20.7 Peer-to-Peer Distributed Search	1020
20.7.1 Peer-to-Peer Networks	1020
20.7.2 The Distributed-Hashing Problem	1021
20.7.3 Centralized Solutions for Distributed Hashing	1022
20.7.4 Chord Circles	1022
20.7.5 Links in Chord Circles	1024
20.7.6 Search Using Finger Tables	1024
20.7.7 Adding New Nodes	1027

20.7.8	When a Peer Leaves the Network	1030
20.7.9	When a Peer Fails	1030
20.7.10	Exercises for Section 20.7	1031
20.8	Summary of Chapter 20	1031
20.9	References for Chapter 20	1033
V	Other Issues in Management of Massive Data	1035
21	Information Integration	1037
21.1	Introduction to Information Integration	1037
21.1.1	Why Information Integration?	1038
21.1.2	The Heterogeneity Problem	1040
21.2	Modes of Information Integration	1041
21.2.1	Federated Database Systems	1042
21.2.2	Data Warehouses	1043
21.2.3	Mediators	1046
21.2.4	Exercises for Section 21.2	1048
21.3	Wrappers in Mediator-Based Systems	1049
21.3.1	Templates for Query Patterns	1050
21.3.2	Wrapper Generators	1051
21.3.3	Filters	1052
21.3.4	Other Operations at the Wrapper	1053
21.3.5	Exercises for Section 21.3	1054
21.4	Capability-Based Optimization	1056
21.4.1	The Problem of Limited Source Capabilities	1056
21.4.2	A Notation for Describing Source Capabilities	1057
21.4.3	Capability-Based Query-Plan Selection	1058
21.4.4	Adding Cost-Based Optimization	1060
21.4.5	Exercises for Section 21.4	1060
21.5	Optimizing Mediator Queries	1061
21.5.1	Simplified Adornment Notation	1061
21.5.2	Obtaining Answers for Subgoals	1062
21.5.3	The Chain Algorithm	1063
21.5.4	Incorporating Union Views at the Mediator	1067
21.5.5	Exercises for Section 21.5	1068
21.6	Local-as-View Mediators	1069
21.6.1	Motivation for LAV Mediators	1069
21.6.2	Terminology for LAV Mediation	1070
21.6.3	Expanding Solutions	1071
21.6.4	Containment of Conjunctive Queries	1073
21.6.5	Why the Containment-Mapping Test Works	1075
21.6.6	Finding Solutions to a Mediator Query	1076
21.6.7	Why the LMSS Theorem Holds	1077
21.6.8	Exercises for Section 21.6	1078

21.7 Entity Resolution	1078
21.7.1 Deciding Whether Records Represent a Common Entity	1079
21.7.2 Merging Similar Records	1081
21.7.3 Useful Properties of Similarity and Merge Functions	1082
21.7.4 The R-Swoosh Algorithm for ICAR Records	1083
21.7.5 Why R-Swoosh Works	1086
21.7.6 Other Approaches to Entity Resolution	1086
21.7.7 Exercises for Section 21.7	1087
21.8 Summary of Chapter 21	1089
21.9 References for Chapter 21	1091
22 Data Mining	1093
22.1 Frequent-Itemset Mining	1093
22.1.1 The Market-Basket Model	1094
22.1.2 Basic Definitions	1095
22.1.3 Association Rules	1097
22.1.4 The Computation Model for Frequent Itemsets	1098
22.1.5 Exercises for Section 22.1	1099
22.2 Algorithms for Finding Frequent Itemsets	1100
22.2.1 The Distribution of Frequent Itemsets	1100
22.2.2 The Naive Algorithm for Finding Frequent Itemsets	1101
22.2.3 The A-Priori Algorithm	1102
22.2.4 Implementation of the A-Priori Algorithm	1104
22.2.5 Making Better Use of Main Memory	1105
22.2.6 When to Use the PCY Algorithm	1106
22.2.7 The Multistage Algorithm	1107
22.2.8 Exercises for Section 22.2	1109
22.3 Finding Similar Items	1110
22.3.1 The Jaccard Measure of Similarity	1110
22.3.2 Applications of Jaccard Similarity	1110
22.3.3 Minhashing	1112
22.3.4 Minhashing and Jaccard Distance	1113
22.3.5 Why Minhashing Works	1113
22.3.6 Implementing Minhashing	1114
22.3.7 Exercises for Section 22.3	1115
22.4 Locality-Sensitive Hashing	1116
22.4.1 Entity Resolution as an Example of LSH	1117
22.4.2 Locality-Sensitive Hashing of Signatures	1118
22.4.3 Combining Minhashing and Locality-Sensitive Hashing	1121
22.4.4 Exercises for Section 22.4	1122
22.5 Clustering of Large-Scale Data	1123
22.5.1 Applications of Clustering	1123
22.5.2 Distance Measures	1125
22.5.3 Agglomerative Clustering	1128
22.5.4 <i>k</i> -Means Algorithms	1130

22.5.5	k -Means for Large-Scale Data	1132
22.5.6	Processing a Memory Load of Points	1133
22.5.7	Exercises for Section 22.5	1136
22.6	Summary of Chapter 22	1137
22.7	References for Chapter 22	1139
23	Database Systems and the Internet	1141
23.1	The Architecture of a Search Engine	1141
23.1.1	Components of a Search Engine	1142
23.1.2	Web Crawlers	1143
23.1.3	Query Processing in Search Engines	1146
23.1.4	Ranking Pages	1146
23.2	PageRank for Identifying Important Pages	1147
23.2.1	The Intuition Behind PageRank	1147
23.2.2	Recursive Formulation of PageRank — First Try	1148
23.2.3	Spider Traps and Dead Ends	1150
23.2.4	PageRank Accounting for Spider Traps and Dead Ends	1153
23.2.5	Exercises for Section 23.2	1154
23.3	Topic-Specific PageRank	1156
23.3.1	Teleport Sets	1156
23.3.2	Calculating A Topic-Specific PageRank	1158
23.3.3	Link Spam	1159
23.3.4	Topic-Specific PageRank and Link Spam	1160
23.3.5	Exercises for Section 23.3	1161
23.4	Data Streams	1161
23.4.1	Data-Stream-Management Systems	1162
23.4.2	Stream Applications	1163
23.4.3	A Data-Stream Data Model	1164
23.4.4	Converting Streams Into Relations	1165
23.4.5	Converting Relations Into Streams	1166
23.4.6	Exercises for Section 23.4	1168
23.5	Data Mining of Streams	1169
23.5.1	Motivation	1169
23.5.2	Counting Bits	1171
23.5.3	Counting the Number of Distinct Elements	1175
23.5.4	Exercises for Section 23.5	1176
23.6	Summary of Chapter 23	1177
23.7	References for Chapter 23	1179
	Index	1183

DATABASE SYSTEMS

The Complete Book

Chapter 1

The Worlds of Database Systems

Databases today are essential to every business. Whenever you visit a major Web site — Google, Yahoo!, Amazon.com, or thousands of smaller sites that provide information — there is a database behind the scenes serving up the information you request. Corporations maintain all their important records in databases. Databases are likewise found at the core of many scientific investigations. They represent the data gathered by astronomers, by investigators of the human genome, and by biochemists exploring properties of proteins, among many other scientific activities.

The power of databases comes from a body of knowledge and technology that has developed over several decades and is embodied in specialized software called a *database management system*, or *DBMS*, or more colloquially a “database system.” A DBMS is a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time, safely. These systems are among the most complex types of software available. In this book, we shall learn how to design databases, how to write programs in the various languages associated with a DBMS, and how to implement the DBMS itself.

1.1 The Evolution of Database Systems

What is a database? In essence a database is nothing more than a collection of information that exists over a long period of time, often many years. In common parlance, the term *database* refers to a collection of data that is managed by a DBMS. The DBMS is expected to:

1. Allow users to create new databases and specify their *schemas* (logical structure of the data), using a specialized *data-definition language*.

2. Give users the ability to *query* the data (a “query” is database lingo for a question about the data) and modify the data, using an appropriate language, often called a *query language* or *data-manipulation language*.
3. Support the storage of very large amounts of data — many terabytes or more — over a long period of time, allowing efficient access to the data for queries and database modifications.
4. Enable *durability*, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.
5. Control access to data from many users at once, without allowing unexpected interactions among users (called *isolation*) and without actions on the data to be performed partially but not completely (called *atomicity*).

1.1.1 Early Database Management Systems

The first commercial database management systems appeared in the late 1960's. These systems evolved from file systems, which provide some of item (3) above; file systems store data over a long period of time, and they allow the storage of large amounts of data. However, file systems do not generally guarantee that data cannot be lost if it is not backed up, and they don't support efficient access to data items whose location in a particular file is not known.

Further, file systems do not directly support item (2), a query language for the data in files. Their support for (1) — a schema for the data — is limited to the creation of directory structures for files. Item (4) is not always supported by file systems; you can lose data that has not been backed up. Finally, file systems do not satisfy (5). While they allow concurrent access to files by several users or processes, a file system generally will not prevent situations such as two users modifying the same file at about the same time, so the changes made by one user fail to appear in the file.

The first important applications of DBMS's were ones where data was composed of many small items, and many queries or modifications were made. Examples of these applications are:

1. Banking systems: maintaining accounts and making sure that system failures do not cause money to disappear.
2. Airline reservation systems: these, like banking systems, require assurance that data will not be lost, and they must accept very large volumes of small actions by customers.
3. Corporate record keeping: employment and tax records, inventories, sales records, and a great variety of other types of information, much of it critical.

The early DBMS's required the programmer to visualize data much as it was stored. These database systems used several different data models for

describing the structure of the information in a database, chief among them the “hierarchical” or tree-based model and the graph-based “network” model. The latter was standardized in the late 1960’s through a report of CODASYL (Committee on Data Systems and Languages).¹

A problem with these early models and systems was that they did not support high-level query languages. For example, the CODASYL query language had statements that allowed the user to jump from data element to data element, through a graph of pointers among these elements. There was considerable effort needed to write such programs, even for very simple queries.

1.1.2 Relational Database Systems

Following a famous paper written by Ted Codd in 1970,² database systems changed significantly. Codd proposed that database systems should present the user with a view of data organized as tables called *relations*. Behind the scenes, there might be a complex data structure that allowed rapid response to a variety of queries. But, unlike the programmers for earlier database systems, the programmer of a relational system would not be concerned with the storage structure. Queries could be expressed in a very high-level language, which greatly increased the efficiency of database programmers. We shall cover the relational model of database systems throughout most of this book. SQL (“Structured Query Language”), the most important query language based on the relational model, is covered extensively.

By 1990, relational database systems were the norm. Yet the database field continues to evolve, and new issues and approaches to the management of data surface regularly. Object-oriented features have infiltrated the relational model. Some of the largest databases are organized rather differently from those using relational methodology. In the balance of this section, we shall consider some of the modern trends in database systems.

1.1.3 Smaller and Smaller Systems

Originally, DBMS’s were large, expensive software systems running on large computers. The size was necessary, because to store a gigabyte of data required a large computer system. Today, hundreds of gigabytes fit on a single disk, and it is quite feasible to run a DBMS on a personal computer. Thus, database systems based on the relational model have become available for even very small machines, and they are beginning to appear as a common tool for computer applications, much as spreadsheets and word processors did before them.

Another important trend is the use of documents, often tagged using XML (eXtensible Modeling Language). Large collections of small documents can

¹CODASYL Data Base Task Group April 1971 Report, ACM, New York.

²Codd, E. F., “A relational model for large shared data banks,” *Comm. ACM*, 13:6, pp. 377–387, 1970.

serve as a database, and the methods of querying and manipulating them are different from those used in relational systems.

1.1.4 Bigger and Bigger Systems

On the other hand, a gigabyte is not that much data any more. Corporate databases routinely store terabytes (10^{12} bytes). Yet there are many databases that store petabytes (10^{15} bytes) of data and serve it all to users. Some important examples:

1. Google holds petabytes of data gleaned from its crawl of the Web. This data is not held in a traditional DBMS, but in specialized structures optimized for search-engine queries.
2. Satellites send down petabytes of information for storage in specialized systems.
3. A picture is actually worth way more than a thousand words. You can store 1000 words in five or six thousand bytes. Storing a picture typically takes much more space. Repositories such as Flickr store millions of pictures and support search of those pictures. Even a database like Amazon's has millions of pictures of products to serve.
4. And if still pictures consume space, movies consume much more. An hour of video requires at least a gigabyte. Sites such as YouTube hold hundreds of thousands, or millions, of movies and make them available easily.
5. Peer-to-peer file-sharing systems use large networks of conventional computers to store and distribute data of various kinds. Although each node in the network may only store a few hundred gigabytes, together the database they embody is enormous.

1.1.5 Information Integration

To a great extent, the old problem of building and maintaining databases has become one of *information integration*: joining the information contained in many related databases into a whole. For example, a large company has many divisions. Each division may have built its own database of products or employee records independently of other divisions. Perhaps some of these divisions used to be independent companies, which naturally had their own way of doing things. These divisions may use different DBMS's and different structures for information. They may use different terms to mean the same thing or the same term to mean different things. To make matters worse, the existence of legacy applications using each of these databases makes it almost impossible to scrap them, ever.

As a result, it has become necessary with increasing frequency to build structures on top of existing databases, with the goal of integrating the information

distributed among them. One popular approach is the creation of *data warehouses*, where information from many legacy databases is copied periodically, with the appropriate translation, to a central database. Another approach is the implementation of a mediator, or “middleware,” whose function is to support an integrated model of the data of the various databases, while translating between this model and the actual models used by each database.

1.2 Overview of a Database Management System

In Fig. 1.1 we see an outline of a complete DBMS. Single boxes represent system components, while double boxes represent in-memory data structures. The solid lines indicate control and data flow, while dashed lines indicate data flow only. Since the diagram is complicated, we shall consider the details in several stages. First, at the top, we suggest that there are two distinct sources of commands to the DBMS:

1. Conventional users and application programs that ask for data or modify data.
2. A *database administrator*: a person or persons responsible for the structure or *schema* of the database.

1.2.1 Data-Definition Language Commands

The second kind of command is the simpler to process, and we show its trail beginning at the upper right side of Fig. 1.1. For example, the database administrator, or *DBA*, for a university registrar's database might decide that there should be a table or relation with columns for a student, a course the student has taken, and a grade for that student in that course. The DBA might also decide that the only allowable grades are A, B, C, D, and F. This structure and constraint information is all part of the schema of the database. It is shown in Fig. 1.1 as entered by the DBA, who needs special authority to execute schema-altering commands, since these can have profound effects on the database. These schema-altering data-definition language (DDL) commands are parsed by a DDL processor and passed to the execution engine, which then goes through the index/file/record manager to alter the *metadata*, that is, the schema information for the database.

1.2.2 Overview of Query Processing

The great majority of interactions with the DBMS follow the path on the left side of Fig. 1.1. A user or an application program initiates some action, using the data-manipulation language (DML). This command does not affect the schema of the database, but may affect the content of the database (if the

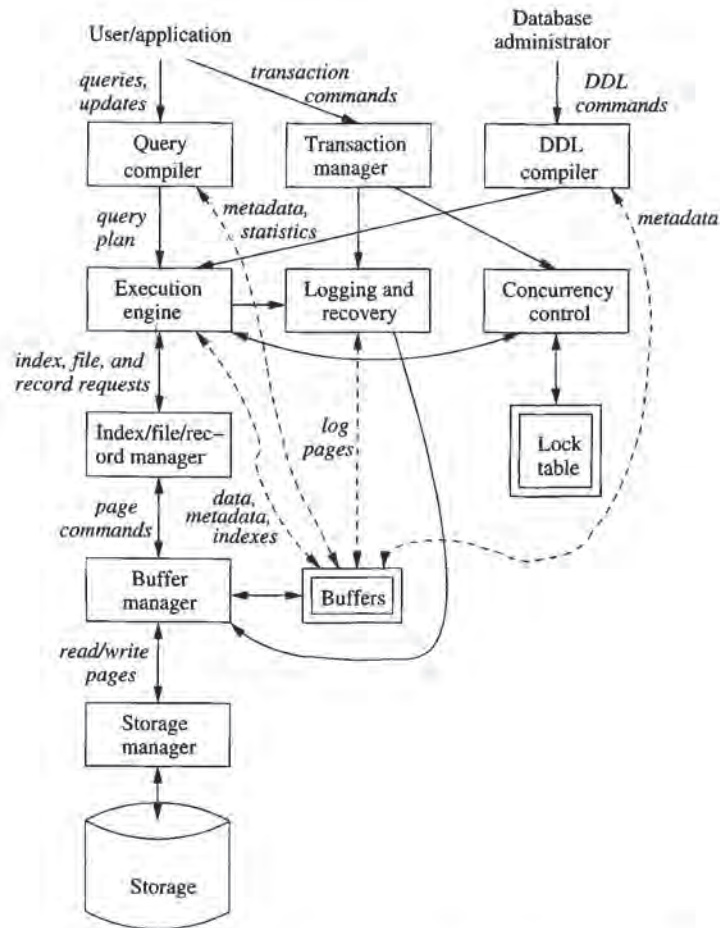


Figure 1.1: Database management system components

action is a modification command) or will extract data from the database (if the action is a query). DML statements are handled by two separate subsystems, as follows.

Answering the Query

The query is parsed and optimized by a *query compiler*. The resulting *query plan*, or sequence of actions the DBMS will perform to answer the query, is passed to the *execution engine*. The execution engine issues a sequence of requests for small pieces of data, typically records or tuples of a relation, to a resource manager that knows about *data files* (holding relations), the format and size of records in those files, and *index files*, which help find elements of data files quickly.

The requests for data are passed to the *buffer manager*. The buffer manager's task is to bring appropriate portions of the data from secondary storage (disk) where it is kept permanently, to the main-memory buffers. Normally, the page or "disk block" is the unit of transfer between buffers and disk.

The buffer manager communicates with a storage manager to get data from disk. The storage manager might involve operating-system commands, but more typically, the DBMS issues commands directly to the disk controller.

Transaction Processing

Queries and other DML actions are grouped into *transactions*, which are units that must be executed atomically and in isolation from one another. Any query or modification action can be a transaction by itself. In addition, the execution of transactions must be *durable*, meaning that the effect of any completed transaction must be preserved even if the system fails in some way right after completion of the transaction. We divide the transaction processor into two major parts:

1. A *concurrency-control manager*, or *scheduler*, responsible for assuring atomicity and isolation of transactions, and
2. A *logging and recovery manager*, responsible for the durability of transactions.

1.2.3 Storage and Buffer Management

The data of a database normally resides in secondary storage; in today's computer systems "secondary storage" generally means magnetic disk. However, to perform any useful operation on data, that data must be in main memory. It is the job of the *storage manager* to control the placement of data on disk and its movement between disk and main memory.

In a simple database system, the storage manager might be nothing more than the file system of the underlying operating system. However, for efficiency

purposes, DBMS's normally control storage on the disk directly, at least under some circumstances. The *storage manager* keeps track of the location of files on the disk and obtains the block or blocks containing a file on request from the buffer manager.

The *buffer manager* is responsible for partitioning the available main memory into *buffers*, which are page-sized regions into which disk blocks can be transferred. Thus, all DBMS components that need information from the disk will interact with the buffers and the buffer manager, either directly or through the execution engine. The kinds of information that various components may need include:

1. *Data*: the contents of the database itself.
2. *Metadata*: the database schema that describes the structure of, and constraints on, the database.
3. *Log Records*: information about recent changes to the database; these support durability of the database.
4. *Statistics*: information gathered and stored by the DBMS about data properties such as the sizes of, and values in, various relations or other components of the database.
5. *Indexes*: data structures that support efficient access to the data.

1.2.4 Transaction Processing

It is normal to group one or more database operations into a *transaction*, which is a unit of work that must be executed atomically and in apparent isolation from other transactions. In addition, a DBMS offers the guarantee of durability: that the work of a completed transaction will never be lost. The *transaction manager* therefore accepts *transaction commands* from an application, which tell the transaction manager when transactions begin and end, as well as information about the expectations of the application (some may not wish to require atomicity, for example). The transaction processor performs the following tasks:

1. *Logging*: In order to assure durability, every change in the database is logged separately on disk. The *log manager* follows one of several policies designed to assure that no matter when a system failure or “crash” occurs, a *recovery manager* will be able to examine the log of changes and restore the database to some consistent state. The log manager initially writes the log in buffers and negotiates with the buffer manager to make sure that buffers are written to disk (where data can survive a crash) at appropriate times.
2. *Concurrency control*: Transactions must appear to execute in isolation. But in most systems, there will in truth be many transactions executing

The ACID Properties of Transactions

Properly implemented transactions are commonly said to meet the “ACID test,” where:

- “A” stands for “atomicity,” the all-or-nothing execution of transactions.
- “I” stands for “isolation,” the fact that each transaction must appear to be executed as if no other transaction is executing at the same time.
- “D” stands for “durability,” the condition that the effect on the database of a transaction must never be lost, once the transaction has completed.

The remaining letter, “C,” stands for “consistency.” That is, all databases have consistency constraints, or expectations about relationships among data elements (e.g., account balances may not be negative after a transaction finishes). Transactions are expected to preserve the consistency of the database.

at once. Thus, the scheduler (concurrency-control manager) must assure that the individual actions of multiple transactions are executed in such an order that the net effect is the same as if the transactions had in fact executed in their entirety, one-at-a-time. A typical scheduler does its work by maintaining *locks* on certain pieces of the database. These locks prevent two transactions from accessing the same piece of data in ways that interact badly. Locks are generally stored in a main-memory *lock table*, as suggested by Fig. 1.1. The scheduler affects the execution of queries and other database operations by forbidding the execution engine from accessing locked parts of the database.

3. *Deadlock resolution*: As transactions compete for resources through the locks that the scheduler grants, they can get into a situation where none can proceed because each needs something another transaction has. The transaction manager has the responsibility to intervene and cancel (“roll-back” or “abort”) one or more transactions to let the others proceed.

1.2.5 The Query Processor

The portion of the DBMS that most affects the performance that the user sees is the *query processor*. In Fig. 1.1 the query processor is represented by two components:

1. The *query compiler*, which translates the query into an internal form called a *query plan*. The latter is a sequence of operations to be performed on the data. Often the operations in a query plan are implementations of “relational algebra” operations, which are discussed in Section 2.4. The query compiler consists of three major units:
 - (a) A *query parser*, which builds a tree structure from the textual form of the query.
 - (b) A *query preprocessor*, which performs semantic checks on the query (e.g., making sure all relations mentioned by the query actually exist), and performing some tree transformations to turn the parse tree into a tree of algebraic operators representing the initial query plan.
 - (c) A *query optimizer*, which transforms the initial query plan into the best available sequence of operations on the actual data.

The query compiler uses metadata and statistics about the data to decide which sequence of operations is likely to be the fastest. For example, the existence of an *index*, which is a specialized data structure that facilitates access to data, given values for one or more components of that data, can make one plan much faster than another.

2. The *execution engine*, which has the responsibility for executing each of the steps in the chosen query plan. The execution engine interacts with most of the other components of the DBMS, either directly or through the buffers. It must get the data from the database into buffers in order to manipulate that data. It needs to interact with the scheduler to avoid accessing data that is locked, and with the log manager to make sure that all database changes are properly logged.

1.3 Outline of Database-System Studies

We divide the study of databases into five parts. This section is an outline of what to expect in each of these units.

Part I: Relational Database Modeling

The relational model is essential for a study of database systems. After examining the basic concepts, we delve into the theory of relational databases. That study includes *functional dependencies*, a formal way of stating that one kind of data is uniquely determined by another. It also includes *normalization*, the process whereby functional dependencies and other formal dependencies are used to improve the design of a relational database.

We also consider high-level design notations. These mechanisms include the Entity-Relationship (E/R) model, Unified Modeling Language (UML), and Object Definition Language (ODL). Their purpose is to allow informal exploration of design issues before we implement the design using a relational DBMS.

Part II: Relational Database Programming

We then take up the matter of how relational databases are queried and modified. After an introduction to abstract programming languages based on algebra and logic (Relational Algebra and Datalog, respectively), we turn our attention to the standard language for relational databases: SQL. We study both the basics and important special topics, including constraint specifications and triggers (active database elements), indexes and other structures to enhance performance, forming SQL into transactions, and security and privacy of data in SQL.

We also discuss how SQL is used in complete systems. It is typical to combine SQL with a conventional or *host* language and to pass data between the database and the conventional program via SQL calls. We discuss a number of ways to make this connection, including embedded SQL, Persistent Stored Modules (PSM), Call-Level Interface (CLI), Java Database Interconnectivity (JDBC), and PHP.

Part III: Semistructured Data Modeling and Programming

The pervasiveness of the Web has put a premium on the management of hierarchically structured data, because the standards for the Web are based on nested, tagged elements (*semistructured data*). We introduce XML and its schema-defining notations: Document Type Definitions (DTD) and XML Schema. We also examine three query languages for XML: XPATH, XQuery, and Extensible Stylesheet Language Transform (XSLT).

Part IV: Database System Implementation

We begin with a study of *storage management*: how disk-based storage can be organized to allow efficient access to data. We explain the commonly used B-tree, a balanced tree of disk blocks and other specialized schemes for managing multidimensional data.

We then turn our attention to *query processing*. There are two parts to this study. First, we need to learn *query execution*: the algorithms used to implement the operations from which queries are built. Since data is typically on disk, the algorithms are somewhat different from what one would expect were they to study the same problems but assuming that data were in main memory. The second step is *query compiling*. Here, we study how to select an efficient query plan from among all the possible ways in which a given query can be executed.

Then, we study *transaction processing*. There are several threads to follow. One concerns *logging*: maintaining reliable records of what the DBMS is doing, in order to allow *recovery* in the event of a crash. Another thread is *scheduling*: controlling the order of events in transactions to assure the ACID properties. We also consider how to deal with deadlocks, and the modifications to our algorithms that are needed when a transaction is *distributed* over many independent

sites.

Part V: Modern Database System Issues

In this part, we take up a number of the ways in which database-system technology is relevant beyond the realm of conventional, relational DBMS's. We consider how *search engines* work, and the specialized data structures that make their operation possible. We look at information integration, and methodologies for making databases share their data seamlessly. *Data mining* is a study that includes a number of interesting and important algorithms for processing large amounts of data in complex ways. *Data-stream systems* deal with data that arrives at the system continuously, and whose queries are answered continuously and in a timely fashion. *Peer-to-peer systems* present many challenges for management of distributed data held by independent hosts.

1.4 References for Chapter 1

Today, on-line searchable bibliographies cover essentially all recent papers concerning database systems. Thus, in this book, we shall not try to be exhaustive in our citations, but rather shall mention only the papers of historical importance and major secondary sources or useful surveys. A searchable index of database research papers was constructed by Michael Ley [5], and has recently been expanded to include references from many fields. Alf-Christian Achilles maintains a searchable directory of many indexes relevant to the database field [3].

While many prototype implementations of database systems contributed to the technology of the field, two of the most widely known are the System R project at IBM Almaden Research Center [4] and the INGRES project at Berkeley [7]. Each was an early relational system and helped establish this type of system as the dominant database technology. Many of the research papers that shaped the database field are found in [6].

The 2003 “Lowell report” [1] is the most recent in a series of reports on database-system research and directions. It also has references to earlier reports of this type.

You can find more about the theory of database systems than is covered here from [2] and [8].

1. S. Abiteboul et al., “The Lowell database research self-assessment,” *Comm. ACM* 48:5 (2005), pp. 111–118. <http://research.microsoft.com/~gray/lowell/LowellDatabaseResearchSelfAssessment.htm>
2. S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, Reading, MA, 1995.
3. <http://liinwww.ira.uka.de/bibliography/Database>.

4. M. M. Astrahan et al., "System R: a relational approach to database management," *ACM Trans. on Database Systems* 1:2, pp. 97–137, 1976.
5. <http://www.informatik.uni-trier.de/~ley/db/index.html>. A mirror site is found at <http://www.acm.org/sigmod/dblp/db/index.html>.
6. M. Stonebraker and J. M. Hellerstein (eds.), *Readings in Database Systems*, Morgan-Kaufmann, San Francisco, 1998.
7. M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," *ACM Trans. on Database Systems* 1:3, pp. 189–222, 1976.
8. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volumes I and II*, Computer Science Press, New York, 1988, 1989.

Part I

**Relational Database
Modeling**

Chapter 2

The Relational Model of Data

This chapter introduces the most important model of data: the two-dimensional table, or “relation.” We begin with an overview of data models in general. We give the basic terminology for relations and show how the model can be used to represent typical forms of data. We then introduce a portion of the language SQL — that part used to declare relations and their structure. The chapter closes with an introduction to relational algebra. We see how this notation serves as both a query language — the aspect of a data model that enables us to ask questions about the data — and as a constraint language — the aspect of a data model that lets us restrict the data in the database in various ways.

2.1 An Overview of Data Models

The notion of a “data model” is one of the most fundamental in the study of database systems. In this brief summary of the concept, we define some basic terminology and mention the most important data models.

2.1.1 What is a Data Model?

A *data model* is a notation for describing data or information. The description generally consists of three parts:

1. *Structure of the data.* You may be familiar with tools in programming languages such as C or Java for describing the structure of the data used by a program: arrays and structures (“structs”) or objects, for example. The data structures used to implement data in the computer are sometimes referred to, in discussions of database systems, as a *physical data model*, although in fact they are far removed from the gates and electrons that truly serve as the physical implementation of the data. In the database

world, data models are at a somewhat higher level than data structures, and are sometimes referred to as a *conceptual model* to emphasize the difference in level. We shall see examples shortly.

2. *Operations on the data.* In programming languages, operations on the data are generally anything that can be programmed. In database data models, there is usually a limited set of operations that can be performed. We are generally allowed to perform a limited set of *queries* (operations that retrieve information) and *modifications* (operations that change the database). This limitation is not a weakness, but a strength. By limiting operations, it is possible for programmers to describe database operations at a very high level, yet have the database management system implement the operations efficiently. In comparison, it is generally impossible to optimize programs in conventional languages like C, to the extent that an inefficient algorithm (e.g., bubblesort) is replaced by a more efficient one (e.g., quicksort).
3. *Constraints on the data.* Database data models usually have a way to describe limitations on what the data can be. These constraints can range from the simple (e.g., “a day of the week is an integer between 1 and 7” or “a movie has at most one title”) to some very complex limitations that we shall discuss in Sections 7.4 and 7.5.

2.1.2 Important Data Models

Today, the two data models of preeminent importance for database systems are:

1. The relational model, including object-relational extensions.
2. The semistructured-data model, including XML and related standards.

The first, which is present in all commercial database management systems, is the subject of this chapter. The semistructured model, of which XML is the primary manifestation, is an added feature of most relational DBMS's, and appears in a number of other contexts as well. We turn to this data model starting in Chapter 11.

2.1.3 The Relational Model in Brief

The relational model is based on tables, of which Fig. 2.1 is an example. We shall discuss this model beginning in Section 2.2. This relation, or table, describes movies: their title, the year in which they were made, their length in minutes, and the genre of the movie. We show three particular movies, but you should imagine that there are many more rows to this table — one row for each movie ever made, perhaps.

The structure portion of the relational model might appear to resemble an array of structs in C, where the column headers are the field names, and each

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Figure 2.1: An example relation

of the rows represent the values of one struct in the array. However, it must be emphasized that this physical implementation is only one possible way the table could be implemented in physical data structures. In fact, it is not the normal way to represent relations, and a large portion of the study of database systems addresses the right ways to implement such tables. Much of the distinction comes from the scale of relations — they are not normally implemented as main-memory structures, and their proper physical implementation must take into account the need to access relations of very large size that are resident on disk.

The operations normally associated with the relational model form the “relational algebra,” which we discuss beginning in Section 2.4. These operations are table-oriented. As an example, we can ask for all those rows of a relation that have a certain value in a certain column. For example, we can ask of the table in Fig. 2.1 for all the rows where the genre is “comedy.”

The constraint portion of the relational data model will be touched upon briefly in Section 2.5 and covered in more detail in Chapter 7. However, as a brief sample of what kinds of constraints are generally used, we could decide that there is a fixed list of genres for movies, and that the last column of every row must have a value that is on this list. Or we might decide (incorrectly, it turns out) that there could never be two movies with the same title, and constrain the table so that no two rows could have the same string in the first component.

2.1.4 The Semistructured Model in Brief

Semistructured data resembles trees or graphs, rather than tables or arrays. The principal manifestation of this viewpoint today is XML, a way to represent data by hierarchically nested tagged elements. The tags, similar to those used in HTML, define the role played by different pieces of data, much as the column headers do in the relational model. For example, the same data as in Fig. 2.1 might appear in an XML “document” as in Fig. 2.2.

The operations on semistructured data usually involve following paths in the implied tree from an element to one or more of its nested subelements, then to subelements nested within those, and so on. For example, starting at the outer <Movies> element (the entire document in Fig. 2.2), we might move to each of its nested <Movie> elements, each delimited by the tag <Movie> and matching </Movie> tag, and from each <Movie> element to its nested <Genre>

```
<Movies>
  <Movie title="Gone With the Wind">
    <Year>1939</Year>
    <Length>231</Length>
    <Genre>drama</Genre>
  </Movie>
  <Movie title="Star Wars">
    <Year>1977</Year>
    <Length>124</Length>
    <Genre>sciFi</Genre>
  </Movie>
  <Movie title="Wayne's World">
    <Year>1992</Year>
    <Length>95</Length>
    <Genre>comedy</Genre>
  </Movie>
</Movies>
```

Figure 2.2: Movie data as XML

element, to see which movies belong to the “comedy” genre.

Constraints on the structure of data in this model often involve the data type of values associated with a tag. For instance, are the values associated with the `<Length>` tag integers or can they be arbitrary character strings? Other constraints determine which tags can appear nested within which other tags. For example, must each `<Movie>` element have a `<Length>` element nested within it? What other tags, besides those shown in Fig. 2.2 might be used within a `<Movie>` element? Can there be more than one genre for a movie? These and other matters will be taken up in Section 11.2.

2.1.5 Other Data Models

There are many other models that are, or have been, associated with DBMS's. A modern trend is to add object-oriented features to the relational model. There are two effects of object-orientation on relations:

1. Values can have structure, rather than being elementary types such as integer or strings, as they were in Fig. 2.1.
2. Relations can have associated methods.

In a sense, these extensions, called the *object-relational* model, are analogous to the way structs in C were extended to objects in C++. We shall introduce the object-relational model in Section 10.3.

There are even database models of the purely object-oriented kind. In these, the relation is no longer the principal data-structuring concept, but becomes only one option among many structures. We discuss an object-oriented database model in Section 4.9.

There are several other models that were used in some of the earlier DBMS's, but that have now fallen out of use. The *hierarchical model* was, like semistructured data, a tree-oriented model. Its drawback was that unlike more modern models, it really operated at the physical level, which made it impossible for programmers to write code at a conveniently high level. Another such model was the *network model*, which was a graph-oriented, physical-level model. In truth, both the hierarchical model and today's semistructured models, allow full graph structures, and do not limit us strictly to trees. However, the generality of graphs was built directly into the network model, rather than favoring trees as these other models do.

2.1.6 Comparison of Modeling Approaches

Even from our brief example, it appears that semistructured models have more flexibility than relations. This difference becomes even more apparent when we discuss, as we shall, how full graph structures are embedded into tree-like, semistructured models. Nevertheless, the relational model is still preferred in DBMS's, and we should understand why. A brief argument follows.

Because databases are large, efficiency of access to data and efficiency of modifications to that data are of great importance. Also very important is ease of use — the productivity of programmers who use the data. Surprisingly, both goals can be achieved with a model, particularly the relational model, that:

1. Provides a simple, limited approach to structuring data, yet is reasonably versatile, so anything can be modeled.
2. Provides a limited, yet useful, collection of operations on data.

Together, these limitations turn into features. They allow us to implement languages, such as SQL, that enable the programmer to express their wishes at a very high level. A few lines of SQL can do the work of thousands of lines of C, or hundreds of lines of the code that had to be written to access data under earlier models such as network or hierarchical. Yet the short SQL programs, because they use a strongly limited sets of operations, can be optimized to run as fast, or faster than the code written in alternative languages.

2.2 Basics of the Relational Model

The relational model gives us a single way to represent data: as a two-dimensional table called a *relation*. Figure 2.1, which we copy here as Fig. 2.3, is an example of a relation, which we shall call *Movies*. The rows each represent a

movie, and the columns each represent a property of movies. In this section, we shall introduce the most important terminology regarding relations, and illustrate them with the *Movies* relation.

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Figure 2.3: The relation *Movies*

2.2.1 Attributes

The columns of a relation are named by *attributes*; in Fig. 2.3 the attributes are *title*, *year*, *length*, and *genre*. Attributes appear at the tops of the columns. Usually, an attribute describes the meaning of entries in the column below. For instance, the column with attribute *length* holds the length, in minutes, of each movie.

2.2.2 Schemas

The name of a relation and the set of attributes for a relation is called the *schema* for that relation. We show the schema for the relation with the relation name followed by a parenthesized list of its attributes. Thus, the schema for relation *Movies* of Fig. 2.3 is

Movies(*title*, *year*, *length*, *genre*)

The attributes in a relation schema are a set, not a list. However, in order to talk about relations we often must specify a “standard” order for the attributes. Thus, whenever we introduce a relation schema with a list of attributes, as above, we shall take this ordering to be the standard order whenever we display the relation or any of its rows.

In the relational model, a database consists of one or more relations. The set of schemas for the relations of a database is called a *relational database schema*, or just a *database schema*.

2.2.3 Tuples

The rows of a relation, other than the header row containing the attribute names, are called *tuples*. A tuple has one *component* for each attribute of the relation. For instance, the first of the three tuples in Fig. 2.3 has the four components *Gone With the Wind*, 1939, 231, and *drama* for attributes *title*, *year*, *length*, and *genre*, respectively. When we wish to write a tuple

Conventions for Relations and Attributes

We shall generally follow the convention that relation names begin with a capital letter, and attribute names begin with a lower-case letter. However, later in this book we shall talk of relations in the abstract, where the names of attributes do not matter. In that case, we shall use single capital letters for both relations and attributes, e.g., $R(A, B, C)$ for a generic relation with three attributes.

in isolation, not as part of a relation, we normally use commas to separate components, and we use parentheses to surround the tuple. For example,

(Gone With the Wind, 1939, 231, drama)

is the first tuple of Fig. 2.3. Notice that when a tuple appears in isolation, the attributes do not appear, so some indication of the relation to which the tuple belongs must be given. We shall always use the order in which the attributes were listed in the relation schema.

2.2.4 Domains

The relational model requires that each component of each tuple be atomic; that is, it must be of some elementary type such as integer or string. It is not permitted for a value to be a record structure, set, list, array, or any other type that reasonably can have its values broken into smaller components.

It is further assumed that associated with each attribute of a relation is a *domain*, that is, a particular elementary type. The components of any tuple of the relation must have, in each component, a value that belongs to the domain of the corresponding column. For example, tuples of the *Movies* relation of Fig. 2.3 must have a first component that is a string, second and third components that are integers, and a fourth component whose value is a string.

It is possible to include the domain, or data type, for each attribute in a relation schema. We shall do so by appending a colon and a type after attributes. For example, we could represent the schema for the *Movies* relation as:

Movies(title:string, year:integer, length:integer, genre:string)

2.2.5 Equivalent Representations of a Relation

Relations are sets of tuples, not lists of tuples. Thus the order in which the tuples of a relation are presented is immaterial. For example, we can list the three tuples of Fig. 2.3 in any of their six possible orders, and the relation is “the same” as Fig. 2.3.

Moreover, we can reorder the attributes of the relation as we choose, without changing the relation. However, when we reorder the relation schema, we must be careful to remember that the attributes are column headers. Thus, when we change the order of the attributes, we also change the order of their columns. When the columns move, the components of tuples change their order as well. The result is that each tuple has its components permuted in the same way as the attributes are permuted.

For example, Fig. 2.4 shows one of the many relations that could be obtained from Fig. 2.3 by permuting rows and columns. These two relations are considered “the same.” More precisely, these two tables are different presentations of the same relation.

<i>year</i>	<i>genre</i>	<i>title</i>	<i>length</i>
1977	sciFi	Star Wars	124
1992	comedy	Wayne’s World	95
1939	drama	Gone With the Wind	231

Figure 2.4: Another presentation of the relation *Movies*

2.2.6 Relation Instances

A relation about movies is not static; rather, relations change over time. We expect to insert tuples for new movies, as these appear. We also expect changes to existing tuples if we get revised or corrected information about a movie, and perhaps deletion of tuples for movies that are expelled from the database for some reason.

It is less common for the schema of a relation to change. However, there are situations where we might want to add or delete attributes. Schema changes, while possible in commercial database systems, can be very expensive, because each of perhaps millions of tuples needs to be rewritten to add or delete components. Also, if we add an attribute, it may be difficult or even impossible to generate appropriate values for the new component in the existing tuples.

We shall call a set of tuples for a given relation an *instance* of that relation. For example, the three tuples shown in Fig. 2.3 form an instance of relation *Movies*. Presumably, the relation *Movies* has changed over time and will continue to change over time. For instance, in 1990, *Movies* did not contain the tuple for *Wayne’s World*. However, a conventional database system maintains only one version of any relation: the set of tuples that are in the relation “now.” This instance of the relation is called the *current instance*.¹

¹Databases that maintain historical versions of data as it existed in past times are called *temporal databases*.

2.2.7 Keys of Relations

There are many constraints on relations that the relational model allows us to place on database schemas. We shall defer much of the discussion of constraints until Chapter 7. However, one kind of constraint is so fundamental that we shall introduce it here: *key* constraints. A set of attributes forms a *key* for a relation if we do not allow two tuples in a relation instance to have the same values in all the attributes of the key.

Example 2.1: We can declare that the relation *Movies* has a key consisting of the two attributes *title* and *year*. That is, we don't believe there could ever be two movies that had both the same title and the same year. Notice that *title* by itself does not form a key, since sometimes "remakes" of a movie appear. For example, there are three movies named *King Kong*, each made in a different year. It should also be obvious that *year* by itself is not a key, since there are usually many movies made in the same year. □

We indicate the attribute or attributes that form a key for a relation by underlining the key attribute(s). For instance, the *Movies* relation could have its schema written as:

Movies(title, year, length, genre)

Remember that the statement that a set of attributes forms a key for a relation is a statement about all possible instances of the relation, not a statement about a single instance. For example, looking only at the tiny relation of Fig. 2.3, we might imagine that *genre* by itself forms a key, since we do not see two tuples that agree on the value of their *genre* components. However, we can easily imagine that if the relation instance contained more movies, there would be many dramas, many comedies, and so on. Thus, there would be distinct tuples that agreed on the *genre* component. As a consequence, it would be incorrect to assert that *genre* is a key for the relation *Movies*.

While we might be sure that *title* and *year* can serve as a key for *Movies*, many real-world databases use artificial keys, doubting that it is safe to make any assumption about the values of attributes outside their control. For example, companies generally assign employee ID's to all employees, and these ID's are carefully chosen to be unique numbers. One purpose of these ID's is to make sure that in the company database each employee can be distinguished from all others, even if there are several employees with the same name. Thus, the employee-ID attribute can serve as a key for a relation about employees.

In US corporations, it is normal for every employee to have a Social-Security number. If the database has an attribute that is the Social-Security number, then this attribute can also serve as a key for employees. Note that there is nothing wrong with there being several choices of key, as there would be for employees having both employee ID's and Social-Security numbers.

The idea of creating an attribute whose purpose is to serve as a key is quite widespread. In addition to employee ID's, we find student ID's to distinguish

students in a university. We find drivers' license numbers and automobile registration numbers to distinguish drivers and automobiles, respectively. You undoubtedly can find more examples of attributes created for the primary purpose of serving as keys.

```

Movies(
    title:string,
    year:integer,
    length:integer,
    genre:string,
    studioName:string,
    producerC#:integer
)
MovieStar(
    name:string,
    address:string,
    gender:char,
    birthdate:date
)
StarsIn(
    movieTitle:string,
    movieYear:integer,
    starName:string
)
MovieExec(
    name:string,
    address:string,
    cert#:integer,
    netWorth:integer
)
Studio(
    name:string,
    address:string,
    presC#:integer
)

```

Figure 2.5: Example database schema about movies

2.2.8 An Example Database Schema

We shall close this section with an example of a complete database schema. The topic is movies, and it builds on the relation *Movies* that has appeared so far in examples. The database schema is shown in Fig. 2.5. Here are the things we need to know to understand the intention of this schema.

Movies

This relation is an extension of the example relation we have been discussing so far. Remember that its key is `title` and `year` together. We have added two new attributes; `studioName` tells us the studio that owns the movie, and `producerC#` is an integer that represents the producer of the movie in a way that we shall discuss when we talk about the relation `MovieExec` below.

MovieStar

This relation tells us something about stars. The key is `name`, the name of the movie star. It is not usual to assume names of persons are unique and therefore suitable as a key. However, movie stars are different; one would never take a name that some other movie star had used. Thus, we shall use the convenient fiction that movie-star names are unique. A more conventional approach would be to invent a serial number of some sort, like social-security numbers, so that we could assign each individual a unique number and use that attribute as the key. We take that approach for movie executives, as we shall see. Another interesting point about the `MovieStar` relation is that we see two new data types. The gender can be a single character, M or F. Also, `birthdate` is of type "date," which might be a character string of a special form.

StarsIn

This relation connects movies to the stars of that movie, and likewise connects a star to the movies in which they appeared. Notice that movies are represented by the key for `Movies` — the title and year — although we have chosen different attribute names to emphasize that attributes `movieTitle` and `movieYear` represent the movie. Likewise, stars are represented by the key for `MovieStar`, with the attribute called `starName`. Finally, notice that all three attributes are necessary to form a key. It is perfectly reasonable to suppose that relation `StarsIn` could have two distinct tuples that agree in any two of the three attributes. For instance, a star might appear in two movies in one year, giving rise to two tuples that agreed in `movieYear` and `starName`, but disagreed in `movieTitle`.

MovieExec

This relation tells us about movie executives. It contains their name, address, and networth as data about the executive. However, for a key we have invented "certificate numbers" for all movie executives, including producers (as appear in the relation `Movies`) and studio presidents (as appear in the relation `Studio`, below). These are integers; a different one is assigned to each executive.

<i>acctNo</i>	<i>type</i>	<i>balance</i>
12345	savings	12000
23456	checking	1000
34567	savings	25

The relation **Accounts**

<i>firstName</i>	<i>lastName</i>	<i>idNo</i>	<i>account</i>
Robbie	Banks	901-222	12345
Lena	Hand	805-333	12345
Lena	Hand	805-333	23456

The relation **Customers**

Figure 2.6: Two relations of a banking database

Studio

This relation tells about movie studios. We rely on no two studios having the same name, and therefore use **name** as the key. The other attributes are the address of the studio and the certificate number for the president of the studio. We assume that the studio president is surely a movie executive and therefore appears in **MovieExec**.

2.2.9 Exercises for Section 2.2

Exercise 2.2.1: In Fig. 2.6 are instances of two relations that might constitute part of a banking database. Indicate the following:

- The attributes of each relation.
- The tuples of each relation.
- The components of one tuple from each relation.
- The relation schema for each relation.
- The database schema.
- A suitable domain for each attribute.
- Another equivalent way to present each relation.

Exercise 2.2.2: In Section 2.2.7 we suggested that there are many examples of attributes that are created for the purpose of serving as keys of relations. Give some additional examples.

!! Exercise 2.2.3: How many different ways (considering orders of tuples and attributes) are there to represent a relation instance if that instance has:

- a) Three attributes and three tuples, like the relation *Accounts* of Fig. 2.6?
- b) Four attributes and five tuples?
- c) n attributes and m tuples?

2.3 Defining a Relation Schema in SQL

SQL (pronounced “sequel”) is the principal language used to describe and manipulate relational databases. There is a current standard for SQL, called SQL-99. Most commercial database management systems implement something similar, but not identical to, the standard. There are two aspects to SQL:

1. The *Data-Definition* sublanguage for declaring database schemas and
2. The *Data-Manipulation* sublanguage for *querying* (asking questions about) databases and for modifying the database.

The distinction between these two sublanguages is found in most languages; e.g., C or Java have portions that declare data and other portions that are executable code. These correspond to data-definition and data-manipulation, respectively.

In this section we shall begin a discussion of the data-definition portion of SQL. There is more on the subject in Chapter 7, especially the matter of constraints on data. The data-manipulation portion is covered extensively in Chapter 6.

2.3.1 Relations in SQL

SQL makes a distinction between three kinds of relations:

1. *Stored relations*, which are called *tables*. These are the kind of relation we deal with ordinarily — a relation that exists in the database and that can be modified by changing its tuples, as well as queried.
2. *Views*, which are relations defined by a computation. These relations are not stored, but are constructed, in whole or in part, when needed. They are the subject of Section 8.1.

3. Temporary tables, which are constructed by the SQL language processor when it performs its job of executing queries and data modifications. These relations are then thrown away and not stored.

In this section, we shall learn how to declare tables. We do not treat the declaration and definition of views here, and temporary tables are never declared. The SQL `CREATE TABLE` statement declares the schema for a stored relation. It gives a name for the table, its attributes, and their data types. It also allows us to declare a key, or even several keys, for a relation. There are many other features to the `CREATE TABLE` statement, including many forms of constraints that can be declared, and the declaration of *indexes* (data structures that speed up many operations on the table) but we shall leave those for the appropriate time.

2.3.2 Data Types

To begin, let us introduce the primitive data types that are supported by SQL systems. All attributes must have a data type.

1. Character strings of fixed or varying length. The type `CHAR(n)` denotes a fixed-length string of up to n characters. `VARCHAR(n)` also denotes a string of up to n characters. The difference is implementation-dependent; typically `CHAR` implies that short strings are padded to make n characters, while `VARCHAR` implies that an endmarker or string-length is used. SQL permits reasonable coercions between values of character-string types. Normally, a string is padded by trailing blanks if it becomes the value of a component that is a fixed-length string of greater length. For example, the string `'foo'`,² if it became the value of a component for an attribute of type `CHAR(5)`, would assume the value `'foo '` (with two blanks following the second o).
2. Bit strings of fixed or varying length. These strings are analogous to fixed and varying-length character strings, but their values are strings of bits rather than characters. The type `BIT(n)` denotes bit strings of length n , while `BIT VARYING(n)` denotes bit strings of length up to n .
3. The type `BOOLEAN` denotes an attribute whose value is logical. The possible values of such an attribute are `TRUE`, `FALSE`, and — although it would surprise George Boole — `UNKNOWN`.
4. The type `INT` or `INTEGER` (these names are synonyms) denotes typical integer values. The type `SHORTINT` also denotes integers, but the number of bits permitted may be less, depending on the implementation (as with the types `int` and `short int` in C).

²Notice that in SQL, strings are surrounded by single-quotes, not double-quotes as in many other programming languages.

Dates and Times in SQL

Different SQL implementations may provide many different representations for dates and times, but the following is the SQL standard representation. A date value is the keyword `DATE` followed by a quoted string of a special form. For example, `DATE '1948-05-14'` follows the required form. The first four characters are digits representing the year. Then come a hyphen and two digits representing the month. Finally there is another hyphen and two digits representing the day. Note that single-digit months and days are padded with a leading 0.

A time value is the keyword `TIME` and a quoted string. This string has two digits for the hour, on the military (24-hour) clock. Then come a colon, two digits for the minute, another colon, and two digits for the second. If fractions of a second are desired, we may continue with a decimal point and as many significant digits as we like. For instance, `TIME '15:00:02.5'` represents the time at which all students will have left a class that ends at 3 PM: two and a half seconds past three o'clock.

5. Floating-point numbers can be represented in a variety of ways. We may use the type `FLOAT` or `REAL` (these are synonyms) for typical floating-point numbers. A higher precision can be obtained with the type `DOUBLE PRECISION`; again the distinction between these types is as in C. SQL also has types that are real numbers with a fixed decimal point. For example, `DECIMAL(n,d)` allows values that consist of n decimal digits, with the decimal point assumed to be d positions from the right. Thus, 0123.45 is a possible value of type `DECIMAL(6,2)`. `NUMERIC` is almost a synonym for `DECIMAL`, although there are possible implementation-dependent differences.
6. Dates and times can be represented by the data types `DATE` and `TIME`, respectively (see the box on "Dates and Times in SQL"). These values are essentially character strings of a special form. We may, in fact, coerce dates and times to string types, and we may do the reverse if the string "makes sense" as a date or time.

2.3.3 Simple Table Declarations

The simplest form of declaration of a relation schema consists of the keywords `CREATE TABLE` followed by the name of the relation and a parenthesized, comma-separated list of the attribute names and their types.

Example 2.2: The relation `Movies` with the schema given in Fig. 2.5 can be declared as in Fig. 2.7. The title is declared as a string of (up to) 100 characters.

```

CREATE TABLE Movies (
    title      CHAR(100),
    year       INT,
    length     INT,
    genre      CHAR(10),
    studioName CHAR(30),
    producerC# INT
);

```

Figure 2.7: SQL declaration of the table *Movies*

The year and length attributes are each integers, and the genre is a string of (up to) 10 characters. The decision to allow up to 100 characters for a title is arbitrary, but we don't want to limit the lengths of titles too strongly, or long titles would be truncated to fit. We have assumed that 10 characters are enough to represent a genre of movie; again, that is an arbitrary choice, one we could regret if we had a genre with a long name. Likewise, we have chosen 30 characters as sufficient for the studio name. The certificate number for the producer of the movie is another integer. □

Example 2.3: Figure 2.8 is a SQL declaration of the relation *MovieStar* from Fig. 2.5. It illustrates some new options for data types. The name of this table is *MovieStar*, and it has four attributes. The first two attributes, *name* and *address*, have each been declared to be character strings. However, with the *name*, we have made the decision to use a fixed-length string of 30 characters, padding a name out with blanks at the end if necessary and truncating a name to 30 characters if it is longer. In contrast, we have declared addresses to be variable-length character strings of up to 255 characters.³ It is not clear that these two choices are the best possible, but we use them to illustrate the two major kinds of string data types.

```

CREATE TABLE MovieStar (
    name      CHAR(30),
    address   VARCHAR(255),
    gender    CHAR(1),
    birthdate DATE
);

```

Figure 2.8: Declaring the relation schema for the *MovieStar* relation

³The number 255 is not the result of some weird notion of what typical addresses look like. A single byte can store integers between 0 and 255, so it is possible to represent a varying-length character string of up to 255 bytes by a single byte for the count of characters plus the bytes to store the string itself. Commercial systems generally support longer varying-length strings, however.

The `gender` attribute has values that are a single letter, M or F. Thus, we can safely use a single character as the type of this attribute. Finally, the `birthdate` attribute naturally deserves the data type `DATE`. $\square$

2.3.4 Modifying Relation Schemas

We now know how to declare a table. But what if we need to change the schema of the table after it has been in use for a long time and has many tuples in its current instance? We can remove the entire table, including all of its current tuples, or we could change the schema by adding or deleting attributes.

We can delete a relation R by the SQL statement:

```
DROP TABLE R;
```

Relation R is no longer part of the database schema, and we can no longer access any of its tuples.

More frequently than we would drop a relation that is part of a long-lived database, we may need to modify the schema of an existing relation. These modifications are done by a statement that begins with the keywords `ALTER TABLE` and the name of the relation. We then have several options, the most important of which are

1. `ADD` followed by an attribute name and its data type.
2. `DROP` followed by an attribute name.

Example 2.4: Thus, for instance, we could modify the `MovieStar` relation by adding an attribute `phone` with:

```
ALTER TABLE MovieStar ADD phone CHAR(16);
```

As a result, the `MovieStar` schema now has five attributes: the four mentioned in Fig. 2.8 and the attribute `phone`, which is a fixed-length string of 16 bytes. In the actual relation, tuples would all have components for `phone`, but we know of no phone numbers to put there. Thus, the value of each of these components is set to the special *null value*, `NULL`. In Section 2.3.5, we shall see how it is possible to choose another “default” value to be used instead of `NULL` for unknown values.

As another example, the `ALTER TABLE` statement:

```
ALTER TABLE MovieStar DROP birthdate;
```

deletes the `birthdate` attribute. As a result, the schema for `MovieStar` no longer has that attribute, and all tuples of the current `MovieStar` instance have the component for `birthdate` deleted. $\square$

2.3.5 Default Values

When we create or modify tuples, we sometimes do not have values for all components. For instance, we mentioned in Example 2.4 that when we add a column to a relation schema, the existing tuples do not have a known value, and it was suggested that NULL could be used in place of a “real” value. However, there are times when we would prefer to use another choice of *default* value, the value that appears in a column if no other value is known.

In general, any place we declare an attribute and its data type, we may add the keyword DEFAULT and an appropriate value. That value is either NULL or a constant. Certain other values that are provided by the system, such as the current time, may also be options.

Example 2.5: Let us consider Example 2.3. We might wish to use the character ? as the default for an unknown gender, and we might also wish to use the earliest possible date, DATE '0000-00-00' for an unknown birthdate. We could replace the declarations of gender and birthdate in Fig. 2.8 by:

```
gender CHAR(1) DEFAULT '?',
birthdate DATE DEFAULT DATE '0000-00-00'
```

As another example, we could have declared the default value for new attribute phone to be 'unlisted' when we added this attribute in Example 2.4. In that case,

```
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

would be the appropriate ALTER TABLE statement. □

2.3.6 Declaring Keys

There are two ways to declare an attribute or set of attributes to be a key in the CREATE TABLE statement that defines a stored relation.

1. We may declare one attribute to be a key when that attribute is listed in the relation schema.
2. We may add to the list of items declared in the schema (which so far have only been attributes) an additional declaration that says a particular attribute or set of attributes forms the key.

If the key consists of more than one attribute, we have to use method (2). If the key is a single attribute, either method may be used.

There are two declarations that may be used to indicate keyness:

- a) PRIMARY KEY, or
- b) UNIQUE.

The effect of declaring a set of attributes S to be a key for relation R either using `PRIMARY KEY` or `UNIQUE` is the following:

- Two tuples in R cannot agree on all of the attributes in set S , unless one of them is `NULL`. Any attempt to insert or update a tuple that violates this rule causes the DBMS to reject the action that caused the violation.

In addition, if `PRIMARY KEY` is used, then attributes in S are not allowed to have `NULL` as a value for their components. Again, any attempt to violate this rule is rejected by the system. `NULL` is permitted if the set S is declared `UNIQUE`, however. A DBMS may make other distinctions between the two terms, if it wishes.

Example 2.6: Let us reconsider the schema for relation `MovieStar`. Since no star would use the name of another star, we shall assume that `name` by itself forms a key for this relation. Thus, we can add this fact to the line declaring `name`. Figure 2.9 is a revision of Fig. 2.8 that reflects this change. We could also substitute `UNIQUE` for `PRIMARY KEY` in this declaration. If we did so, then two or more tuples could have `NULL` as the value of `name`, but there could be no other duplicate values for this attribute.

```
CREATE TABLE MovieStar (  
    name CHAR(30) PRIMARY KEY,  
    address VARCHAR(255),  
    gender CHAR(1),  
    birthdate DATE  
);
```

Figure 2.9: Making `name` the key

Alternatively, we can use a separate definition of the key. The resulting schema declaration would look like Fig. 2.10. Again, `UNIQUE` could replace `PRIMARY KEY`. □

```
CREATE TABLE MovieStar (  
    name CHAR(30),  
    address VARCHAR(255),  
    gender CHAR(1),  
    birthdate DATE,  
    PRIMARY KEY (name)  
);
```

Figure 2.10: A separate declaration of the key

Example 2.7: In Example 2.6, the form of either Fig. 2.9 or Fig. 2.10 is acceptable, because the key is a single attribute. However, in a situation where the key has more than one attribute, we must use the style of Fig. 2.10. For instance, the relation *Movie*, whose key is the pair of attributes *title* and *year*, must be declared as in Fig. 2.11. However, as usual, *UNIQUE* is an option to replace *PRIMARY KEY*. □

```
CREATE TABLE Movies (
    title      CHAR(100),
    year       INT,
    length     INT,
    genre      CHAR(10),
    studioName CHAR(30),
    producerC# INT,
    PRIMARY KEY (title, year)
);
```

Figure 2.11: Making *title* and *year* be the key of *Movies*

2.3.7 Exercises for Section 2.3

Exercise 2.3.1: In this exercise we introduce one of our running examples of a relational database schema. The database schema consists of four relations, whose schemas are:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

The *Product* relation gives the manufacturer, model number and type (PC, laptop, or printer) of various products. We assume for convenience that model numbers are unique over all manufacturers and product types; that assumption is not realistic, and a real database would include a code for the manufacturer as part of the model number. The *PC* relation gives for each model number that is a PC the speed (of the processor, in gigahertz), the amount of RAM (in megabytes), the size of the hard disk (in gigabytes), and the price. The *Laptop* relation is similar, except that the screen size (in inches) is also included. The *Printer* relation records for each printer model whether the printer produces color output (true, if so), the process type (laser or ink-jet, typically), and the price.

Write the following declarations:

- a) A suitable schema for relation *Product*.

- b) A suitable schema for relation `PC`.
- c) A suitable schema for relation `Laptop`.
- d) A suitable schema for relation `Printer`.
- e) An alteration to your `Printer` schema from (d) to delete the attribute `color`.
- f) An alteration to your `Laptop` schema from (c) to add the attribute `od` (optical-disk type, e.g., `cd` or `dvd`). Let the default value for this attribute be `'none'` if the laptop does not have an optical disk.

Exercise 2.3.2: This exercise introduces another running example, concerning World War II capital ships. It involves the following relations:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

Ships are built in “classes” from the same design, and the class is usually named for the first ship of that class. The relation `Classes` records the name of the class, the type (`'bb'` for battleship or `'bc'` for battlecruiser), the country that built the ship, the number of main guns, the bore (diameter of the gun barrel, in inches) of the main guns, and the displacement (weight, in tons). Relation `Ships` records the name of the ship, the name of its class, and the year in which the ship was launched. Relation `Battles` gives the name and date of battles involving these ships, and relation `Outcomes` gives the result (sunk, damaged, or ok) for each ship in each battle.

Write the following declarations:

- a) A suitable schema for relation `Classes`.
- b) A suitable schema for relation `Ships`.
- c) A suitable schema for relation `Battles`.
- d) A suitable schema for relation `Outcomes`.
- e) An alteration to your `Classes` relation from (a) to delete the attribute `bore`.
- f) An alteration to your `Ships` relation from (b) to include the attribute `yard` giving the shipyard where the ship was built.

2.4 An Algebraic Query Language

In this section, we introduce the data-manipulation aspect of the relational model. Recall that a data model is not just structure; it needs a way to query the data and to modify the data. To begin our study of operations on relations, we shall learn about a special algebra, called *relational algebra*, that consists of some simple but powerful ways to construct new relations from given relations. When the given relations are stored data, then the constructed relations can be answers to queries about this data.

Relational algebra is not used today as a query language in commercial DBMS's, although some of the early prototypes did use this algebra directly. Rather, the "real" query language, SQL, incorporates relational algebra at its center, and many SQL programs are really "syntactically sugared" expressions of relational algebra. Further, when a DBMS processes queries, the first thing that happens to a SQL query is that it gets translated into relational algebra or a very similar internal representation. Thus, there are several good reasons to start out learning this algebra.

2.4.1 Why Do We Need a Special Query Language?

Before introducing the operations of relational algebra, one should ask why, or whether, we need a new kind of programming languages for databases. Won't conventional languages like C or Java suffice to ask and answer any computable question about relations? After all, we can represent a tuple of a relation by a struct (in C) or an object (in Java), and we can represent relations by arrays of these elements.

The surprising answer is that relational algebra is useful because it is *less* powerful than C or Java. That is, there are computations one can perform in any conventional language that one cannot perform in relational algebra. An example is: determine whether the number of tuples in a relation is even or odd. By limiting what we can say or do in our query language, we get two huge rewards — ease of programming and the ability of the compiler to produce highly optimized code — that we discussed in Section 2.1.6.

2.4.2 What is an Algebra?

An algebra, in general, consists of operators and atomic operands. For instance, in the algebra of arithmetic, the atomic operands are variables like x and constants like 15. The operators are the usual arithmetic ones: addition, subtraction, multiplication, and division. Any algebra allows us to build *expressions* by applying operators to atomic operands and/or other expressions of the algebra. Usually, parentheses are needed to group operators and their operands. For instance, in arithmetic we have expressions such as $(x + y) * z$ or $((x + 7)/(y - 3)) + x$.

Relational algebra is another example of an algebra. Its atomic operands are:

1. Variables that stand for relations.
2. Constants, which are finite relations.

We shall next see the operators of relational algebra.

2.4.3 Overview of Relational Algebra

The operations of the traditional relational algebra fall into four broad classes:

- a) The usual set operations — union, intersection, and difference — applied to relations.
- b) Operations that remove parts of a relation: “selection” eliminates some rows (tuples), and “projection” eliminates some columns.
- c) Operations that combine the tuples of two relations, including “Cartesian product,” which pairs the tuples of two relations in all possible ways, and various kinds of “join” operations, which selectively pair tuples from two relations.
- d) An operation called “renaming” that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.

We generally shall refer to expressions of relational algebra as *queries*.

2.4.4 Set Operations on Relations

The three most common operations on sets are union, intersection, and difference. We assume the reader is familiar with these operations, which are defined as follows on arbitrary sets R and S :

- $R \cup S$, the *union* of R and S , is the set of elements that are in R or S or both. An element appears only once in the union even if it is present in both R and S .
- $R \cap S$, the *intersection* of R and S , is the set of elements that are in both R and S .
- $R - S$, the *difference* of R and S , is the set of elements that are in R but not in S . Note that $R - S$ is different from $S - R$; the latter is the set of elements that are in S but not in R .

When we apply these operations to relations, we need to put some conditions on R and S :

1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S .
2. Before we compute the set-theoretic union, intersection, or difference of sets of tuples, the columns of R and S must be ordered so that the order of attributes is the same for both relations.

Sometimes we would like to take the union, intersection, or difference of relations that have the same number of attributes, with corresponding domains, but that use different names for their attributes. If so, we may use the renaming operator to be discussed in Section 2.4.11 to change the schema of one or both relations and give them the same set of attributes.

<i>name</i>	<i>address</i>	<i>gender</i>	<i>birthdate</i>
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Relation R

<i>name</i>	<i>address</i>	<i>gender</i>	<i>birthdate</i>
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Relation S

Figure 2.12: Two relations

Example 2.8: Suppose we have the two relations R and S , whose schemas are both that of relation *MovieStar* Section 2.2.8. Current instances of R and S are shown in Fig. 2.12. Then the union $R \cup S$ is

<i>name</i>	<i>address</i>	<i>gender</i>	<i>birthdate</i>
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Note that the two tuples for Carrie Fisher from the two relations appear only once in the result.

The intersection $R \cap S$ is

<i>name</i>	<i>address</i>	<i>gender</i>	<i>birthdate</i>
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Now, only the Carrie Fisher tuple appears, because only it is in both relations.

The difference $R - S$ is

<i>name</i>	<i>address</i>	<i>gender</i>	<i>birthdate</i>
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

That is, the Fisher and Hamill tuples appear in R and thus are candidates for $R - S$. However, the Fisher tuple also appears in S and so is not in $R - S$. $\square$

2.4.5 Projection

The *projection* operator is used to produce from a relation R a new relation that has only some of R 's columns. The value of expression $\pi_{A_1, A_2, \dots, A_n}(R)$ is a relation that has only the columns for attributes $A_1, A_2, \dots, A_n$ of R . The schema for the resulting value is the set of attributes $\{A_1, A_2, \dots, A_n\}$, which we conventionally show in the order listed.

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>producerC#</i>
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890
Wayne's World	1992	95	comedy	Paramount	99999

Figure 2.13: The relation Movies

Example 2.9: Consider the relation Movies with the relation schema described in Section 2.2.8. An instance of this relation is shown in Fig. 2.13. We can project this relation onto the first three attributes with the expression:

$$\pi_{\text{title, year, length}}(\text{Movies})$$

The resulting relation is

<i>title</i>	<i>year</i>	<i>length</i>
Star Wars	1977	124
Galaxy Quest	1999	104
Wayne's World	1992	95

As another example, we can project onto the attribute *genre* with the expression $\pi_{\text{genre}}(\text{Movies})$. The result is the single-column relation

<i>genre</i>
sciFi
comedy

Notice that there are only two tuples in the resulting relation, since the last two tuples of Fig. 2.13 have the same value in their component for attribute *genre*, and in the relational algebra of sets, duplicate tuples are always eliminated. $\square$

A Note About Data Quality :-)

While we have endeavored to make example data as accurate as possible, we have used bogus values for addresses and other personal information about movie stars, in order to protect the privacy of members of the acting profession, many of whom are shy individuals who shun publicity.

2.4.6 Selection

The *selection* operator, applied to a relation R , produces a new relation with a subset of R 's tuples. The tuples in the resulting relation are those that satisfy some condition C that involves the attributes of R . We denote this operation $\sigma_C(R)$. The schema for the resulting relation is the same as R 's schema, and we conventionally show the attributes in the same order as we use for R .

C is a conditional expression of the type with which we are familiar from conventional programming languages; for example, conditional expressions follow the keyword *if* in programming languages such as C or Java. The only difference is that the operands in condition C are either constants or attributes of R . We apply C to each tuple t of R by substituting, for each attribute A appearing in condition C , the component of t for attribute A . If after substituting for each attribute of C the condition C is true, then t is one of the tuples that appear in the result of $\sigma_C(R)$; otherwise t is not in the result.

Example 2.10: Let the relation *Movies* be as in Fig. 2.13. Then the value of expression $\sigma_{length \geq 100}(\text{Movies})$ is

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>producerC#</i>
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890

The first tuple satisfies the condition $length \geq 100$ because when we substitute for *length* the value 124 found in the component of the first tuple for attribute *length*, the condition becomes $124 \geq 100$. The latter condition is true, so we accept the first tuple. The same argument explains why the second tuple of Fig. 2.13 is in the result.

The third tuple has a *length* component 95. Thus, when we substitute for *length* we get the condition $95 \geq 100$, which is false. Hence the last tuple of Fig. 2.13 is not in the result. $\square$

Example 2.11: Suppose we want the set of tuples in the relation *Movies* that represent Fox movies at least 100 minutes long. We can get these tuples with a more complicated condition, involving the AND of two subconditions. The expression is

$$\sigma_{length \geq 100 \text{ AND } studioName = 'Fox'}(\text{Movies})$$

The tuple

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>producerC#</i>
Star Wars	1977	124	sciFi	Fox	12345

is the only one in the resulting relation. $\square$

2.4.7 Cartesian Product

The *Cartesian product* (or *cross-product*, or just *product*) of two sets R and S is the set of pairs that can be formed by choosing the first element of the pair to be any element of R and the second any element of S . This product is denoted $R \times S$. When R and S are relations, the product is essentially the same. However, since the members of R and S are tuples, usually consisting of more than one component, the result of pairing a tuple from R with a tuple from S is a longer tuple, with one component for each of the components of the constituent tuples. By convention, the components from R (the left operand) precede the components from S in the attribute order for the result.

The relation schema for the resulting relation is the union of the schemas for R and S . However, if R and S should happen to have some attributes in common, then we need to invent new names for at least one of each pair of identical attributes. To disambiguate an attribute A that is in the schemas of both R and S , we use $R.A$ for the attribute from R and $S.A$ for the attribute from S .

Example 2.12: For conciseness, let us use an abstract example that illustrates the product operation. Let relations R and S have the schemas and tuples shown in Fig. 2.14(a) and (b). Then the product $R \times S$ consists of the six tuples shown in Fig. 2.14(c). Note how we have paired each of the two tuples of R with each of the three tuples of S . Since B is an attribute of both schemas, we have used $R.B$ and $S.B$ in the schema for $R \times S$. The other attributes are unambiguous, and their names appear in the resulting schema unchanged. $\square$

2.4.8 Natural Joins

More often than we want to take the product of two relations, we find a need to *join* them by pairing only those tuples that match in some way. The simplest sort of match is the *natural join* of two relations R and S , denoted $R \bowtie S$, in which we pair only those tuples from R and S that agree in whatever attributes are common to the schemas of R and S . More precisely, let $A_1, A_2, \dots, A_n$ be all the attributes that are in both the schema of R and the schema of S . Then a tuple r from R and a tuple s from S are successfully paired if and only if r and s agree on each of the attributes $A_1, A_2, \dots, A_n$.

If the tuples r and s are successfully paired in the join $R \bowtie S$, then the result of the pairing is a tuple, called the *joined tuple*, with one component for each of the attributes in the union of the schemas of R and S . The joined tuple

A	B
1	2
3	4

(a) Relation R

B	C	D
2	5	6
4	7	8
10	10	11

(b) Relation S

A	$R.B$	$S.B$	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

(c) Result $R \times S$

Figure 2.14: Two relations and their Cartesian product

agrees with tuple r in each attribute in the schema of R , and it agrees with s in each attribute in the schema of S . Since r and s are successfully paired, the joined tuple is able to agree with both these tuples on the attributes they have in common. The construction of the joined tuple is suggested by Fig. 2.15. However, the order of the attributes need not be that convenient; the attributes of R and S can appear in any order.

Example 2.13: The natural join of the relations R and S from Fig. 2.14(a) and (b) is

A	B	C	D
1	2	5	6
3	4	7	8

The only attribute common to R and S is B . Thus, to pair successfully, tuples need only to agree in their B components. If so, the resulting tuple has components for attributes A (from R), B (from either R or S), C (from S), and D (from S).

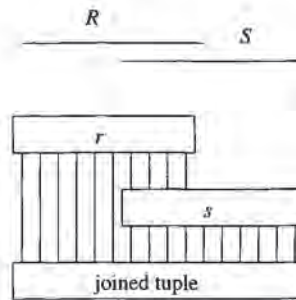


Figure 2.15: Joining tuples

In this example, the first tuple of R successfully pairs with only the first tuple of S ; they share the value 2 on their common attribute B . This pairing yields the first tuple of the result: $(1, 2, 5, 6)$. The second tuple of R pairs successfully only with the second tuple of S , and the pairing yields $(3, 4, 7, 8)$. Note that the third tuple of S does not pair with any tuple of R and thus has no effect on the result of $R \bowtie S$. A tuple that fails to pair with any tuple of the other relation in a join is said to be a *dangling tuple*. $\square$

Example 2.14: The previous example does not illustrate all the possibilities inherent in the natural join operator. For example, no tuple paired successfully with more than one tuple, and there was only one attribute in common to the two relation schemas. In Fig. 2.16 we see two other relations, U and V , that share two attributes between their schemas: B and C . We also show an instance in which one tuple joins with several tuples.

For tuples to pair successfully, they must agree in both the B and C components. Thus, the first tuple of U joins with the first two tuples of V , while the second and third tuples of U join with the third tuple of V . The result of these four pairings is shown in Fig. 2.16(c). $\square$

2.4.9 Theta-Joins

The natural join forces us to pair tuples using one specific condition. While this way, equating shared attributes, is the most common basis on which relations are joined, it is sometimes desirable to pair tuples from two relations on some other basis. For that purpose, we have a related notation called the *theta-join*. Historically, the “theta” refers to an arbitrary condition, which we shall represent by C rather than θ .

The notation for a theta-join of relations R and S based on condition C is $R \bowtie_C S$. The result of this operation is constructed as follows:

1. Take the product of R and S .
2. Select from the product only those tuples that satisfy the condition C .

<i>A</i>	<i>B</i>	<i>C</i>
1	2	3
6	7	8
9	7	8

(a) Relation U

<i>B</i>	<i>C</i>	<i>D</i>
2	3	4
2	3	5
7	8	10

(b) Relation V

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

(c) Result $U \bowtie V$

Figure 2.16: Natural join of relations

As with the product operation, the schema for the result is the union of the schemas of R and S , with “ R .” or “ S .” prefixed to attributes if necessary to indicate from which schema the attribute came.

Example 2.15: Consider the operation $U \bowtie_{A < D} V$, where U and V are the relations from Fig. 2.16(a) and (b). We must consider all nine pairs of tuples, one from each relation, and see whether the A component from the U -tuple is less than the D component of the V -tuple. The first tuple of U , with an A component of 1, successfully pairs with each of the tuples from V . However, the second and third tuples from U , with A components of 6 and 9, respectively, pair successfully with only the last tuple of V . Thus, the result has only five tuples, constructed from the five successful pairings. This relation is shown in Fig. 2.17. $\square$

Notice that the schema for the result in Fig. 2.17 consists of all six attributes, with U and V prefixed to their respective occurrences of attributes B and C to distinguish them. Thus, the theta-join contrasts with natural join, since in the latter common attributes are merged into one copy. Of course it makes sense to

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

Figure 2.17: Result of $U \bowtie_{A < D} V$

do so in the case of the natural join, since tuples don't pair unless they agree in their common attributes. In the case of a theta-join, there is no guarantee that compared attributes will agree in the result, since they may not be compared with $=$.

Example 2.16: Here is a theta-join on the same relations U and V that has a more complex condition:

$$U \bowtie_{A < D \text{ AND } U.B \neq V.B} V$$

That is, we require for successful pairing not only that the A component of the U -tuple be less than the D component of the V -tuple, but that the two tuples disagree on their respective B components. The tuple

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	7	8	10

is the only one to satisfy both conditions, so this relation is the result of the theta-join above. $\square$

2.4.10 Combining Operations to Form Queries

If all we could do was to write single operations on one or two relations as queries, then relational algebra would not be nearly as useful as it is. However, relational algebra, like all algebras, allows us to form expressions of arbitrary complexity by applying operations to the result of other operations.

One can construct expressions of relational algebra by applying operators to subexpressions, using parentheses when necessary to indicate grouping of operands. It is also possible to represent expressions as expression trees; the latter often are easier for us to read, although they are less convenient as a machine-readable notation.

Example 2.17: Suppose we want to know, from our running *Movies* relation, "What are the titles and years of movies made by Fox that are at least 100 minutes long?" One way to compute the answer to this query is:

1. Select those *Movies* tuples that have $length \geq 100$.

2. Select those *Movies* tuples that have *studioName* = 'Fox'.
3. Compute the intersection of (1) and (2).
4. Project the relation from (3) onto attributes *title* and *year*.

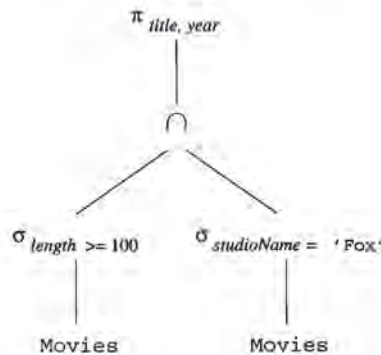


Figure 2.18: Expression tree for a relational algebra expression

In Fig. 2.18 we see the above steps represented as an expression tree. Expression trees are evaluated bottom-up by applying the operator at an interior node to the arguments, which are the results of its children. By proceeding bottom-up, we know that the arguments will be available when we need them. The two selection nodes correspond to steps (1) and (2). The intersection node corresponds to step (3), and the projection node is step (4).

Alternatively, we could represent the same expression in a conventional, linear notation, with parentheses. The formula

$$\pi_{title, year} \left(\sigma_{length \geq 100}(\text{Movies}) \cap \sigma_{studioName = 'Fox'}(\text{Movies}) \right)$$

represents the same expression.

Incidentally, there is often more than one relational algebra expression that represents the same computation. For instance, the above query could also be written by replacing the intersection by logical AND within a single selection operation. That is,

$$\pi_{title, year} \left(\sigma_{length \geq 100 \text{ AND } studioName = 'Fox'}(\text{Movies}) \right)$$

is an equivalent form of the query. $\square$

Equivalent Expressions and Query Optimization

All database systems have a query-answering system, and many of them are based on a language that is similar in expressive power to relational algebra. Thus, the query asked by a user may have many *equivalent expressions* (expressions that produce the same answer whenever they are given the same relations as operands), and some of these may be much more quickly evaluated. An important job of the query “optimizer” discussed briefly in Section 1.2.5 is to replace one expression of relational algebra by an equivalent expression that is more efficiently evaluated.

2.4.11 Naming and Renaming

In order to control the names of the attributes used for relations that are constructed by applying relational-algebra operations, it is often convenient to use an operator that explicitly renames relations. We shall use the operator $\rho_{S(A_1, A_2, \dots, A_n)}(R)$ to rename a relation R . The resulting relation has exactly the same tuples as R , but the name of the relation is S . Moreover, the attributes of the result relation S are named $A_1, A_2, \dots, A_n$, in order from the left. If we only want to change the name of the relation to S and leave the attributes as they are in R , we can just say $\rho_S(R)$.

Example 2.18: In Example 2.12 we took the product of two relations R and S from Fig. 2.14(a) and (b) and used the convention that when an attribute appears in both operands, it is renamed by prefixing the relation name to it. Suppose, however, that we do not wish to call the two versions of B by names $R.B$ and $S.B$; rather we want to continue to use the name B for the attribute that comes from R , and we want to use X as the name of the attribute B coming from S . We can rename the attributes of S so the first is called X . The result of the expression $\rho_{S(X, C, D)}(S)$ is a relation named S that looks just like the relation S from Fig. 2.14, but its first column has attribute X instead of B .

A	B	X	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Figure 2.19: $R \times \rho_{S(X, C, D)}(S)$

When we take the product of R with this new relation, there is no conflict of names among the attributes, so no further renaming is done. That is, the result of the expression $R \times \rho_{S(X,C,D)}(S)$ is the relation $R \times S$ from Fig. 2.14(c), except that the five columns are labeled A , B , X , C , and D , from the left. This relation is shown in Fig. 2.19.

As an alternative, we could take the product without renaming, as we did in Example 2.12, and then rename the result. The expression

$$\rho_{RS(A,B,X,C,D)}(R \times S)$$

yields the same relation as in Fig. 2.19, with the same set of attributes. But this relation has a name, RS , while the result relation in Fig. 2.19 has no name. $\square$

2.4.12 Relationships Among Operations

Some of the operations that we have described in Section 2.4 can be expressed in terms of other relational-algebra operations. For example, intersection can be expressed in terms of set difference:

$$R \cap S = R - (R - S)$$

That is, if R and S are any two relations with the same schema, the intersection of R and S can be computed by first subtracting S from R to form a relation T consisting of all those tuples in R but not S . We then subtract T from R , leaving only those tuples of R that are also in S .

The two forms of join are also expressible in terms of other operations. Theta-join can be expressed by product and selection:

$$R \bowtie_C S = \sigma_C(R \times S)$$

The natural join of R and S can be expressed by starting with the product $R \times S$. We then apply the selection operator with a condition C of the form

$$R.A_1 = S.A_1 \text{ AND } R.A_2 = S.A_2 \text{ AND } \cdots \text{ AND } R.A_n = S.A_n$$

where $A_1, A_2, \dots, A_n$ are all the attributes appearing in the schemas of both R and S . Finally, we must project out one copy of each of the equated attributes. Let L be the list of attributes in the schema of R followed by those attributes in the schema of S that are not also in the schema of R . Then

$$R \bowtie S = \pi_L(\sigma_C(R \times S))$$

Example 2.19: The natural join of the relations U and V from Fig. 2.16 can be written in terms of product, selection, and projection as:

$$\pi_{A,U,B,U,C,D}(\sigma_{U.B=V.B \text{ AND } U.C=V.C}(U \times V))$$

That is, we take the product $U \times V$. Then we select for equality between each pair of attributes with the same name — B and C in this example. Finally, we project onto all the attributes except one of the B 's and one of the C 's; we have chosen to eliminate the attributes of V whose names also appear in the schema of U .

For another example, the theta-join of Example 2.16 can be written

$$\sigma_{A < D \text{ AND } U.B \neq V.B}(U \times V)$$

That is, we take the product of the relations U and V and then apply the condition that appeared in the theta-join. $\square$

The rewriting rules mentioned in this section are the only “redundancies” among the operations that we have introduced. The six remaining operations — union, difference, selection, projection, product, and renaming — form an independent set, none of which can be written in terms of the other five.

2.4.13 A Linear Notation for Algebraic Expressions

In Section 2.4.10 we used an expression tree to represent a complex expression of relational algebra. An alternative is to invent names for the temporary relations that correspond to the interior nodes of the tree and write a sequence of assignments that create a value for each. The order of the assignments is flexible, as long as the children of a node N have had their values created before we attempt to create the value for N itself.

The notation we shall use for assignment statements is:

1. A relation name and parenthesized list of attributes for that relation. The name **Answer** will be used conventionally for the result of the final step; i.e., the name of the relation at the root of the expression tree.
2. The assignment symbol $:=$.
3. Any algebraic expression on the right. We can choose to use only one operator per assignment, in which case each interior node of the tree gets its own assignment statement. However, it is also permissible to combine several algebraic operations in one right side, if it is convenient to do so.

Example 2.20: Consider the tree of Fig. 2.18. One possible sequence of assignments to evaluate this expression is:

```
R(t,y,l,i,s,p) :=  $\sigma_{length \geq 100}$ (Movies)
S(t,y,l,i,s,p) :=  $\sigma_{studioName = 'Fox'}$ (Movies)
T(t,y,l,i,s,p) :=  $R \cap S$ 
Answer(title, year) :=  $\pi_{t,y}$ (T)
```

The first step computes the relation of the interior node labeled $\sigma_{length \geq 100}$ in Fig. 2.18, and the second step computes the node labeled $\sigma_{studioName='Fox'}$. Notice that we get renaming “for free,” since we can use any attributes and relation name we wish for the left side of an assignment. The last two steps compute the intersection and the projection in the obvious way.

It is also permissible to combine some of the steps. For instance, we could combine the last two steps and write:

```
R(t,y,l,i,s,p) :=  $\sigma_{length \geq 100}$ (Movies)
S(t,y,l,i,s,p) :=  $\sigma_{studioName='Fox'}$ (Movies)
Answer(title, year) :=  $\pi_{t,y}(R \cap S)$ 
```

We could even substitute for R and S in the last line and write the entire expression in one line. $\square$

2.4.14 Exercises for Section 2.4

Exercise 2.4.1: This exercise builds upon the products schema of Exercise 2.3.1. Recall that the database schema consists of four relations, whose schemas are:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

Some sample data for the relation *Product* is shown in Fig. 2.20. Sample data for the other three relations is shown in Fig. 2.21. Manufacturers and model numbers have been “sanitized,” but the data is typical of products on sale at the beginning of 2007.

Write expressions of relational algebra to answer the following queries. You may use the linear notation of Section 2.4.13 if you wish. For the data of Figs. 2.20 and 2.21, show the result of your query. However, your answer should work for arbitrary data, not just the data of these figures.

- a) What PC models have a speed of at least 3.00?
- b) Which manufacturers make laptops with a hard disk of at least 100GB?
- c) Find the model number and price of all products (of any type) made by manufacturer *B*.
- d) Find the model numbers of all color laser printers.
- e) Find those manufacturers that sell Laptops, but not PC's.
- f) Find those hard-disk sizes that occur in two or more PC's.

<i>maker</i>	<i>model</i>	<i>type</i>
A	1001	pc
A	1002	pc
A	1003	pc
A	2004	laptop
A	2005	laptop
A	2006	laptop
B	1004	pc
B	1005	pc
B	1006	pc
B	2007	laptop
C	1007	pc
D	1008	pc
D	1009	pc
D	1010	pc
D	3004	printer
D	3005	printer
E	1011	pc
E	1012	pc
E	1013	pc
E	2001	laptop
E	2002	laptop
E	2003	laptop
E	3001	printer
E	3002	printer
E	3003	printer
F	2008	laptop
F	2009	laptop
G	2010	laptop
H	3006	printer
H	3007	printer

Figure 2.20: Sample data for Product

<i>model</i>	<i>speed</i>	<i>ram</i>	<i>hd</i>	<i>price</i>
1001	2.66	1024	250	2114
1002	2.10	512	250	995
1003	1.42	512	80	478
1004	2.80	1024	250	649
1005	3.20	512	250	630
1006	3.20	1024	320	1049
1007	2.20	1024	200	510
1008	2.20	2048	250	770
1009	2.00	1024	250	650
1010	2.80	2048	300	770
1011	1.86	2048	160	959
1012	2.80	1024	160	649
1013	3.06	512	80	529

(a) Sample data for relation PC

<i>model</i>	<i>speed</i>	<i>ram</i>	<i>hd</i>	<i>screen</i>	<i>price</i>
2001	2.00	2048	240	20.1	3673
2002	1.73	1024	80	17.0	949
2003	1.80	512	60	15.4	549
2004	2.00	512	60	13.3	1150
2005	2.16	1024	120	17.0	2500
2006	2.00	2048	80	15.4	1700
2007	1.83	1024	120	13.3	1429
2008	1.60	1024	100	15.4	900
2009	1.60	512	80	14.1	680
2010	2.00	2048	160	15.4	2300

(b) Sample data for relation Laptop

<i>model</i>	<i>color</i>	<i>type</i>	<i>price</i>
3001	true	ink-jet	99
3002	false	laser	239
3003	true	laser	899
3004	true	ink-jet	120
3005	false	laser	120
3006	true	ink-jet	100
3007	true	laser	200

(c) Sample data for relation Printer

Figure 2.21: Sample data for relations of Exercise 2.4.1

- ! g) Find those pairs of PC models that have both the same speed and RAM. A pair should be listed only once; e.g., list (i, j) but not (j, i) .
- !! h) Find those manufacturers of at least two different computers (PC's or laptops) with speeds of at least 2.80.
- !! i) Find the manufacturer(s) of the computer (PC or laptop) with the highest available speed.
- !! j) Find the manufacturers of PC's with at least three different speeds.
- !! k) Find the manufacturers who sell exactly three different models of PC.

Exercise 2.4.2: Draw expression trees for each of your expressions of Exercise 2.4.1.

Exercise 2.4.3: This exercise builds upon Exercise 2.3.2 concerning World War II capital ships. Recall it involves the following relations:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

Figures 2.22 and 2.23 give some sample data for these four relations.⁴ Note that, unlike the data for Exercise 2.4.1, there are some “dangling tuples” in this data, e.g., ships mentioned in *Outcomes* that are not mentioned in *Ships*.

Write expressions of relational algebra to answer the following queries. You may use the linear notation of Section 2.4.13 if you wish. For the data of Figs. 2.22 and 2.23, show the result of your query. However, your answer should work for arbitrary data, not just the data of these figures.

- a) Give the class names and countries of the classes that carried guns of at least 16-inch bore.
- b) Find the ships launched prior to 1921.
- c) Find the ships sunk in the battle of the Denmark Strait.
- d) The treaty of Washington in 1921 prohibited capital ships heavier than 35,000 tons. List the ships that violated the treaty of Washington.
- e) List the name, displacement, and number of guns of the ships engaged in the battle of Guadalcanal.
- f) List all the capital ships mentioned in the database. (Remember that all these ships may not appear in the *Ships* relation.)

⁴Source: J. N. Westwood, *Fighting Ships of World War II*, Follett Publishing, Chicago, 1975 and R. C. Stern, *US Battleships in Action*, Squadron/Signal Publications, Carrollton, TX, 1980.

<i>class</i>	<i>type</i>	<i>country</i>	<i>numGuns</i>	<i>bore</i>	<i>displacement</i>
Bismarck	bb	Germany	8	15	42000
Iowa	bb	USA	9	16	46000
Kongo	bc	Japan	8	14	32000
North Carolina	bb	USA	9	16	37000
Renown	bc	Gt. Britain	6	15	32000
Revenge	bb	Gt. Britain	8	15	29000
Tennessee	bb	USA	12	14	32000
Yamato	bb	Japan	9	18	65000

(a) Sample data for relation Classes

<i>name</i>	<i>date</i>
Denmark Strait	5/24-27/41
Guadalcanal	11/15/42
North Cape	12/26/43
Surigao Strait	10/25/44

(b) Sample data for relation Battles

<i>ship</i>	<i>battle</i>	<i>result</i>
Arizona	Pearl Harbor	sunk
Bismarck	Denmark Strait	sunk
California	Surigao Strait	ok
Duke of York	North Cape	ok
Fuso	Surigao Strait	sunk
Hood	Denmark Strait	sunk
King George V	Denmark Strait	ok
Kirishima	Guadalcanal	sunk
Prince of Wales	Denmark Strait	damaged
Rodney	Denmark Strait	ok
Scharnhorst	North Cape	sunk
South Dakota	Guadalcanal	damaged
Tennessee	Surigao Strait	ok
Washington	Guadalcanal	ok
West Virginia	Surigao Strait	ok
Yamashiro	Surigao Strait	sunk

(c) Sample data for relation Outcomes

Figure 2.22: Data for Exercise 2.4.3

<i>name</i>	<i>class</i>	<i>launched</i>
California	Tennessee	1921
Haruna	Kongo	1915
Hiei	Kongo	1914
Iowa	Iowa	1943
Kirishima	Kongo	1915
Kongo	Kongo	1913
Missouri	Iowa	1944
Musashi	Yamato	1942
New Jersey	Iowa	1943
North Carolina	North Carolina	1941
Ramillies	Revenge	1917
Renown	Renown	1916
Repulse	Renown	1916
Resolution	Revenge	1916
Revenge	Revenge	1916
Royal Oak	Revenge	1916
Royal Sovereign	Revenge	1916
Tennessee	Tennessee	1920
Washington	North Carolina	1941
Wisconsin	Iowa	1944
Yamato	Yamato	1941

Figure 2.23: Sample data for relation Ships

- ! g) Find the classes that had only one ship as a member of that class.
- ! h) Find those countries that had both battleships and battlecruisers.
- ! i) Find those ships that “lived to fight another day”; they were damaged in one battle, but later fought in another.

Exercise 2.4.4: Draw expression trees for each of your expressions of Exercise 2.4.3.

Exercise 2.4.5: What is the difference between the natural join $R \bowtie S$ and the theta-join $R \bowtie_C S$ where the condition C is that $R.A = S.A$ for each attribute A appearing in the schemas of both R and S ?

! Exercise 2.4.6: An operator on relations is said to be *monotone* if whenever we add a tuple to one of its arguments, the result contains all the tuples that it contained before adding the tuple, plus perhaps more tuples. Which of the operators described in this section are monotone? For each, either explain why it is monotone or give an example showing it is not.

! **Exercise 2.4.7:** Suppose relations R and S have n tuples and m tuples, respectively. Give the minimum and maximum numbers of tuples that the results of the following expressions can have.

- a) $R \cup S$.
- b) $R \bowtie S$.
- c) $\sigma_C(R) \times S$, for some condition C .
- d) $\pi_L(R) - S$, for some list of attributes L .

! **Exercise 2.4.8:** The *semijoin* of relations R and S , written $R \ltimes S$, is the set of tuples t in R such that there is at least one tuple in S that agrees with t in all attributes that R and S have in common. Give three different expressions of relational algebra that are equivalent to $R \ltimes S$.

! **Exercise 2.4.9:** The *antisemijoin* $R \overline{\ltimes} S$ is the set of tuples t in R that do *not* agree with any tuple of S in the attributes common to R and S . Give an expression of relational algebra equivalent to $R \overline{\ltimes} S$.

!! **Exercise 2.4.10:** Let R be a relation with schema

$$(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$$

and let S be a relation with schema $(B_1, B_2, \dots, B_m)$; that is, the attributes of S are a subset of the attributes of R . The *quotient* of R and S , denoted $R \div S$, is the set of tuples t over attributes $A_1, A_2, \dots, A_n$ (i.e., the attributes of R that are not attributes of S) such that for every tuple s in S , the tuple ts , consisting of the components of t for $A_1, A_2, \dots, A_n$ and the components of s for $B_1, B_2, \dots, B_m$, is a member of R . Give an expression of relational algebra, using the operators we have defined previously in this section, that is equivalent to $R \div S$.

2.5 Constraints on Relations

We now take up the third important aspect of a data model: the ability to restrict the data that may be stored in a database. So far, we have seen only one kind of constraint, the requirement that an attribute or attributes form a key (Section 2.3.6). These and many other kinds of constraints can be expressed in relational algebra. In this section, we show how to express both key constraints and “referential-integrity” constraints; the latter require that a value appearing in one column of one relation also appear in some other column of the same or a different relation. In Chapter 7, we see how SQL database systems can enforce the same sorts of constraints as we can express in relational algebra.

2.5.1 Relational Algebra as a Constraint Language

There are two ways in which we can use expressions of relational algebra to express constraints.

1. If R is an expression of relational algebra, then $R = \emptyset$ is a constraint that says “The value of R must be empty,” or equivalently “There are no tuples in the result of R .”
2. If R and S are expressions of relational algebra, then $R \subseteq S$ is a constraint that says “Every tuple in the result of R must also be in the result of S .” Of course the result of S may contain additional tuples not produced by R .

These ways of expressing constraints are actually equivalent in what they can express, but sometimes one or the other is clearer or more succinct. That is, the constraint $R \subseteq S$ could just as well have been written $R - S = \emptyset$. To see why, notice that if every tuple in R is also in S , then surely $R - S$ is empty. Conversely, if $R - S$ contains no tuples, then every tuple in R must be in S (or else it would be in $R - S$).

On the other hand, a constraint of the first form, $R = \emptyset$, could just as well have been written $R \subseteq \emptyset$. Technically, $\emptyset$ is not an expression of relational algebra, but since there are expressions that evaluate to $\emptyset$, such as $R - R$, there is no harm in using $\emptyset$ as a relational-algebra expression.

In the following sections, we shall see how to express significant constraints in one of these two styles. As we shall see in Chapter 7, it is the first style — equal-to-the-emptyset — that is most commonly used in SQL programming. However, as shown above, we are free to think in terms of set-containment if we wish and later convert our constraint to the equal-to-the-emptyset style.

2.5.2 Referential Integrity Constraints

A common kind of constraint, called a *referential integrity constraint*, asserts that a value appearing in one context also appears in another, related context. For example, in our movies database, should we see a **StarsIn** tuple that has person p in the **starName** component, we would expect that p appears as the name of some star in the **MovieStar** relation. If not, then we would question whether the listed “star” really was a star.

In general, if we have any value v as the component in attribute A of some tuple in one relation R , then because of our design intentions we may expect that v will appear in a particular component (say for attribute B) of some tuple of another relation S . We can express this integrity constraint in relational algebra as $\pi_A(R) \subseteq \pi_B(S)$, or equivalently, $\pi_A(R) - \pi_B(S) = \emptyset$.

Example 2.21: Consider the two relations from our running movie database:

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

We might reasonably assume that the producer of every movie would have to appear in the `MovieExec` relation. If not, there is something wrong, and we would at least want a system implementing a relational database to inform us that we had a movie with a producer of which the database had no knowledge.

To be more precise, the `producerC#` component of each `Movies` tuple must also appear in the `cert#` component of some `MovieExec` tuple. Since executives are uniquely identified by their certificate numbers, we would thus be assured that the movie's producer is found among the movie executives. We can express this constraint by the set-containment

$$\pi_{\text{producerC\#}}(\text{Movies}) \subseteq \pi_{\text{cert\#}}(\text{MovieExec})$$

The value of the expression on the left is the set of all certificate numbers appearing in `producerC#` components of `Movies` tuples. Likewise, the expression on the right's value is the set of all certificates in the `cert#` component of `MovieExec` tuples. Our constraint says that every certificate in the former set must also be in the latter set. $\square$

Example 2.22: We can similarly express a referential integrity constraint where the “value” involved is represented by more than one attribute. For instance, we may want to assert that any movie mentioned in the relation

`StarsIn(movieTitle, movieYear, starName)`

also appears in the relation

`Movies(title, year, length, genre, studioName, producerC#)`

`Movies` are represented in both relations by title-year pairs, because we agreed that one of these attributes alone was not sufficient to identify a movie. The constraint

$$\pi_{\text{movieTitle, movieYear}}(\text{StarsIn}) \subseteq \pi_{\text{title, year}}(\text{Movies})$$

expresses this referential integrity constraint by comparing the title-year pairs produced by projecting both relations onto the appropriate lists of components. $\square$

2.5.3 Key Constraints

The same constraint notation allows us to express far more than referential integrity. Here, we shall see how we can express algebraically the constraint that a certain attribute or set of attributes is a key for a relation.

Example 2.23: Recall that name is the key for relation

`MovieStar(name, address, gender, birthdate)`

That is, no two tuples agree on the name component. We shall express algebraically one of several implications of this constraint: that if two tuples agree on name, then they must also agree on address. Note that in fact these “two” tuples, which agree on the key name, must be the same tuple and therefore certainly agree in all attributes.

The idea is that if we construct all pairs of MovieStar tuples (t_1, t_2) , we must not find a pair that agree in the name component and disagree in the address component. To construct the pairs we use a Cartesian product, and to search for pairs that violate the condition we use a selection. We then assert the constraint by equating the result to $\emptyset$.

To begin, since we are taking the product of a relation with itself, we need to rename at least one copy, in order to have names for the attributes of the product. For succinctness, let us use two new names, MS1 and MS2, to refer to the MovieStar relation. Then the requirement can be expressed by the algebraic constraint:

$$\sigma_{MS1.name=MS2.name \text{ AND } MS1.address \neq MS2.address}(MS1 \times MS2) = \emptyset$$

In the above, MS1 in the product $MS1 \times MS2$ is shorthand for the renaming:

$$\rho_{MS1(name,address,gender,birthdate)}(MovieStar)$$

and MS2 is a similar renaming of MovieStar. $\square$

2.5.4 Additional Constraint Examples

There are many other kinds of constraints that we can express in relational algebra and that are useful for restricting database contents. A large family of constraints involve the permitted values in a context. For example, the fact that each attribute has a type constrains the values of that attribute. Often the constraint is quite straightforward, such as “integers only” or “character strings of length up to 30.” Other times we want the values that may appear in an attribute to be restricted to a small enumerated set of values. Other times, there are complex limitations on the values that may appear. We shall give two examples, one of a simple *domain constraint* for an attribute, and the second a more complicated restriction.

Example 2.24: Suppose we wish to specify that the only legal values for the gender attribute of MovieStar are 'F' and 'M'. We can express this constraint algebraically by:

$$\sigma_{gender \neq 'F' \text{ AND } gender \neq 'M'}(MovieStar) = \emptyset$$

That is, the set of tuples in MovieStar whose gender component is equal to neither 'F' nor 'M' is empty. $\square$

Example 2.25: Suppose we wish to require that one must have a net worth of at least \$10,000,000 to be the president of a movie studio. We can express this constraint algebraically as follows. First, we need to theta-join the two relations

```
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

using the condition that `presC#` from `Studio` and `cert#` from `MovieExec` are equal. That join combines pairs of tuples consisting of a studio and an executive, such that the executive is the president of the studio. If we select from this relation those tuples where the net worth is less than ten million, we have a set that, according to our constraint, must be empty. Thus, we may express the constraint as:

$$\sigma_{\text{netWorth} < 10000000}(\text{Studio} \bowtie_{\text{presC\#} = \text{cert\#}} \text{MovieExec}) = \emptyset$$

An alternative way to express the same constraint is to compare the set of certificates that represent studio presidents with the set of certificates that represent executives with a net worth of at least \$10,000,000; the former must be a subset of the latter. The containment

$$\pi_{\text{presC\#}}(\text{Studio}) \subseteq \pi_{\text{cert\#}}(\sigma_{\text{netWorth} \geq 10000000}(\text{MovieExec}))$$

expresses the above idea. $\square$

2.5.5 Exercises for Section 2.5

Exercise 2.5.1: Express the following constraints about the relations of Exercise 2.3.1, reproduced here:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

You may write your constraints either as containments or by equating an expression to the empty set. For the data of Exercise 2.4.1, indicate any violations to your constraints.

- a) A PC with a processor speed less than 2.00 must not sell for more than \$500.
- b) A laptop with a screen size less than 15.4 inches must have at least a 100 gigabyte hard disk or sell for less than \$1000.
- ! c) No manufacturer of PC's may also make laptops.

- !! d) A manufacturer of a PC must also make a laptop with at least as great a processor speed.
- ! e) If a laptop has a larger main memory than a PC, then the laptop must also have a higher price than the PC.

Exercise 2.5.2: Express the following constraints in relational algebra. The constraints are based on the relations of Exercise 2.3.2:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

You may write your constraints either as containments or by equating an expression to the empty set. For the data of Exercise 2.4.3, indicate any violations to your constraints.

- a) No class of ships may have guns with larger than 16-inch bore.
 - b) If a class of ships has more than 9 guns, then their bore must be no larger than 14 inches.
 - ! c) No class may have more than 2 ships.
 - ! d) No country may have both battleships and battlecruisers.
 - !! e) No ship with more than 9 guns may be in a battle with a ship having fewer than 9 guns that was sunk.
- ! Exercise 2.5.3:** Suppose R and S are two relations. Let C be the referential integrity constraint that says: whenever R has a tuple with some values $v_1, v_2, \dots, v_n$ in particular attributes $A_1, A_2, \dots, A_n$, there must be a tuple of S that has the same values $v_1, v_2, \dots, v_n$ in particular attributes $B_1, B_2, \dots, B_n$. Show how to express constraint C in relational algebra.
- ! Exercise 2.5.4:** Another algebraic way to express a constraint is $E_1 = E_2$, where both E_1 and E_2 are relational-algebra expressions. Can this form of constraint express more than the two forms we discussed in this section?

2.6 Summary of Chapter 2

- ◆ **Data Models:** A data model is a notation for describing the structure of the data in a database, along with the constraints on that data. The data model also normally provides a notation for describing operations on that data: queries and data modifications.

- ◆ *Relational Model*: Relations are tables representing information. Columns are headed by attributes; each attribute has an associated domain, or data type. Rows are called tuples, and a tuple has one component for each attribute of the relation.
- ◆ *Schemas*: A relation name, together with the attributes of that relation and their types, form the relation schema. A collection of relation schemas forms a database schema. Particular data for a relation or collection of relations is called an instance of that relation schema or database schema.
- ◆ *Keys*: An important type of constraint on relations is the assertion that an attribute or set of attributes forms a key for the relation. No two tuples of a relation can agree on all attributes of the key, although they can agree on some of the key attributes.
- ◆ *Semistructured Data Model*: In this model, data is organized in a tree or graph structure. XML is an important example of a semistructured data model.
- ◆ *SQL*: The language SQL is the principal query language for relational database systems. The current standard is called SQL-99. Commercial systems generally vary from this standard but adhere to much of it.
- ◆ *Data Definition*: SQL has statements to declare elements of a database schema. The CREATE TABLE statement allows us to declare the schema for stored relations (called tables), specifying the attributes, their types, default values, and keys.
- ◆ *Altering Schemas*: We can change parts of the database schema with an ALTER statement. These changes include adding and removing attributes from relation schemas and changing the default value associated with an attribute. We may also use a DROP statement to completely eliminate relations or other schema elements.
- ◆ *Relational Algebra*: This algebra underlies most query languages for the relational model. Its principal operators are union, intersection, difference, selection, projection, Cartesian product, natural join, theta-join, and renaming.
- ◆ *Selection and Projection*: The selection operator produces a result consisting of all tuples of the argument relation that satisfy the selection condition. Projection removes undesired columns from the argument relation to produce the result.
- ◆ *Joins*: We join two relations by comparing tuples, one from each relation. In a natural join, we splice together those pairs of tuples that agree on all attributes common to the two relations. In a theta-join, pairs of tuples are concatenated if they meet a selection condition associated with the theta-join.

- ♦ *Constraints in Relational Algebra*: Many common kinds of constraints can be expressed as the containment of one relational algebra expression in another, or as the equality of a relational algebra expression to the empty set.

2.7 References for Chapter 2

The classic paper by Codd on the relational model is [1]. This paper introduces relational algebra, as well. The use of relational algebra to describe constraints is from [2]. References for SQL are given in the bibliographic notes for Chapter 6.

The semistructured data model is from [3]. XML is a standard developed by the World-Wide-Web Consortium. The home page for information about XML is [4].

1. E. F. Codd, "A relational model for large shared data banks," *Comm. ACM* **13**:6, pp. 377–387, 1970.
2. J.-M. Nicolas, "Logic for improving integrity checking in relational databases," *Acta Informatica* **18**:3, pp. 227–253, 1982.
3. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources," *IEEE Intl. Conf. on Data Engineering*, pp. 251–260, March 1995.
4. World-Wide-Web Consortium, <http://www.w3.org/XML/>