

# Preface

At Stanford, we are on the quarter system, and as a result, our introductory database instruction is divided into two courses. The first, CS145, is designed for students who will use database systems but not necessarily take a job implementing a DBMS. It is a prerequisite for CS245, which is the introduction to DBMS implementation. Students wishing to go further in the database field then take CS345 (special topics), CS346 (DBMS implementation project), and CS347 (transaction processing and distributed databases).

Starting in 1997, we published a pair of books. *A First Course in Database Systems* was designed for CS145, and *Database System Implementation* was for CS245 and parts of CS346. Because many schools are on the semester system or combine the two kinds of database instruction into one introductory course, we felt that there was a need to produce the two books as a single volume, which we call *Database Systems: The Complete Book*.

- If you are a student considering buying this book, and anticipate a later study of implementation, you should consider buying *Database Systems: The Complete Book* instead.

However, because many more students need to know how to use database systems than to implement them, we have continued to package the first half of the “complete book” as *A First Course in Database Systems*. In the third edition, we have introduced many new topics and altered the overall viewpoint somewhat. Today, there are two important models for database systems: relational and semistructured (XML). We have decided therefore to downplay object-oriented databases, except in the contexts of design and object-relational systems.

## What’s New in the Third 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 earlier, more extensive treatment of ODL and OQL is available on the book's Web site.

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 indexes and views 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-tiered 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. Most 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.

## Use of the Book

There is adequate material in this volume for a one-semester course on database modeling and programming. For a one-quarter course, you will probably have to omit some of the topics. We regard Chapters 2–7 as the core of the course. The remaining five chapters contain material from which it is safe to select at will, although we believe that every student should get some exposure to the issues of embedding SQL in standard host languages from one of the sections in Chapter 9.

If, as we do in CS145, you give students a substantial database-application design and implementation project, then you may have to reorder the material somewhat, so that SQL instruction occurs earlier. You may wish to defer material such as dependencies, although students need normalization for design.

## Prerequisites

We have used the book at the “mezzanine” level, in a course taken both by undergraduates and 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.

## Gradiance On-Line Exercises

There is an accompanying set of on-line homeworks using a technology developed by Gradiance Corp. Instructors may assign these homeworks to their class, or students not enrolled in a class may enroll in an “omnibus class” that allows them to do the homeworks as a tutorial (without an instructor-created class). Gradiance questions look like ordinary questions, but your solutions are sampled. If you make an incorrect choice you are given specific advice or feedback to help you correct your solution. If your instructor permits, you are allowed to try again, until you get a perfect score.

In addition, the Gradiance package for the 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 only respond correctly to the examples is not sufficient to get credit for the problem.

Gradiance 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.

# Support on the World Wide Web

The book's home page is

<http://www-db.stanford.edu/~ullman/fcdb.html>

You will find errata as we learn of them, and backup materials. We are making available the notes for each offering of CS145 as we teach it, including homeworks, projects and exams. We shall also make available there the sections from the second edition that have been removed from the third.

## Mapping the Second Edition to the Third

Here is a table giving the correspondence between old sections and new.

Old	New	Old	New	Old	New	Old	New	Old	New
<b>1.1</b>	1.1	<b>1.2</b>	1.2	<b>1.3</b>	1.3	<b>2.1</b>	4.1	<b>2.2</b>	4.2
<b>2.3</b>	4.3	<b>2.4</b>	4.4	<b>3.1</b>	2.2	<b>3.2</b>	4.5	<b>3.3</b>	4.6
<b>3.4</b>	3.1	<b>3.5</b>	3.2	<b>3.6</b>	3.3–5	<b>3.7</b>	3.6–7	<b>4.1</b>	Web
<b>4.2</b>	4.9	<b>4.3</b>	4.9	<b>4.4</b>	4.10	<b>4.5</b>	10.3	<b>4.6</b>	11.1
<b>4.7</b>	11.2	<b>5.1</b>	2.2	<b>5.2</b>	2.4	<b>5.3</b>	5.1	<b>5.4</b>	5.2
<b>5.5</b>	2.5	<b>6.1</b>	6.1	<b>6.2</b>	6.2	<b>6.3</b>	6.3	<b>6.4</b>	6.4
<b>6.5</b>	6.5	<b>6.6</b>	2.3	<b>6.7</b>	8.1–2	<b>7.1</b>	7.1	<b>7.2</b>	7.2
<b>7.3</b>	7.3	<b>7.4</b>	7.4–5	<b>8.1</b>	9.3	<b>8.2</b>	9.4	<b>8.3</b>	9.2
<b>8.4</b>	9.5	<b>8.5</b>	9.6	<b>8.6</b>	6.6	<b>8.7</b>	10.1	<b>9.1</b>	Web
<b>9.2</b>	Web	<b>9.3</b>	Web	<b>9.4</b>	10.4	<b>9.5</b>	10.5	<b>10.1</b>	5.3
<b>10.2</b>	5.4	<b>10.3</b>	Web	<b>10.4</b>	10.2				

## 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 Frettloeh, Jarl Friis.

Also John Fry, Chiping Fu, Tracy Fujieda, Prasanna Ganesan, Suzanne Garcia, Mark Gjol, Manish Godara, Seth Goldberg, Jeff Goldblat, Meredith Goldsmith, Luis Gravano, Gerard Guillemette, Himanshu Gupta, Petri Gynther, 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, Seth Katz, Pedram Keyani, Victor Kimeli, Ed Knorr, Yeong-Ping Koh, David Koller, Gyorgy Kovacs, Phillip Koza, Brian Kulman, Bill Labiosa, Sang Ho Lee, Younghan 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, Torbjorn Norbye, Chang-Min Oh, Mehul Patel, Soren Peen, Jian Pei.

Also Xiaobo Peng, Bert Porter, Limbek Reka, Prahash Ramanan, Nisheeth Ranjan, Suzanne Rivoire, Ken Ross, Tim Roughgarten, Mema Roussopoulos, Richard Scherl, Loren Shevitz, June Yoshiko Sison, 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, Mike Wiacek, Kristian Widjaja, Janet Wu, Sundar Yannunachari, Takeshi Yokukawa, Bing Yu, Min-Sig Yun, Torben Zahle, Sandy Zhang. The remaining errors are ours, of course.

J. D. U.  
J. W.  
Stanford, CA  
July, 2007

# Table of Contents

<b>1</b>	<b>The Worlds of Database Systems</b>	<b>1</b>
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

<b>I</b>	<b>Relational Database Modeling</b>	<b>15</b>
<b>2</b>	<b>The Relational Model of Data</b>	<b>17</b>
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	<b>Design Theory for Relational Databases</b>	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

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

<b>4 High-Level Database Models</b>	<b>125</b>
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
<b>4.8</b>	<b>From UML Diagrams to Relations . . . . .</b>	<b>179</b>
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
<b>4.9</b>	<b>Object Definition Language . . . . .</b>	<b>183</b>
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
<b>4.10</b>	<b>From ODL Designs to Relational Designs . . . . .</b>	<b>193</b>
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
<b>4.11</b>	<b>Summary of Chapter 4 . . . . .</b>	<b>200</b>
<b>4.12</b>	<b>References for Chapter 4 . . . . .</b>	<b>202</b>
<b>II</b>	<b>Relational Database Programming</b>	<b>203</b>
<b>5</b>	<b>Algebraic and Logical Query Languages</b>	<b>205</b>
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
<b>6</b>	<b>The Database Language SQL</b>	<b>243</b>
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
<b>7</b>	<b>Constraints and Triggers</b>	<b>311</b>
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	<b>Views and Indexes</b>	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
<b>9</b>	<b>SQL in a Server Environment</b>	<b>369</b>
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

<b>10</b>	<b>Advanced Topics in Relational Databases</b> . . . . .	<b>425</b>
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

<b>11 The Semistructured-Data Model</b> . . . . .	<b>483</b>
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

<b>11.4 XML Schema . . . . .</b>	<b>502</b>
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
<b>11.5 Summary of Chapter 11 . . . . .</b>	<b>514</b>
<b>11.6 References for Chapter 11 . . . . .</b>	<b>515</b>
<b>12 Programming Languages for XML . . . . .</b>	<b>517</b>
<b>12.1 XPath . . . . .</b>	<b>517</b>
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
<b>12.2 XQuery . . . . .</b>	<b>528</b>
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
<b>12.3 Extensible Stylesheet Language . . . . .</b>	<b>544</b>
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
<b>12.4 Summary of Chapter 12 . . . . .</b>	<b>553</b>

12.5 References for Chapter 12 . . . . .	554
<b>Index</b>	<b>555</b>

# 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).<sup>1</sup>

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,<sup>2</sup> 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

---

<sup>1</sup> CODASYL Data Base Task Group April 1971 Report, ACM, New York.

<sup>2</sup> Codd, 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

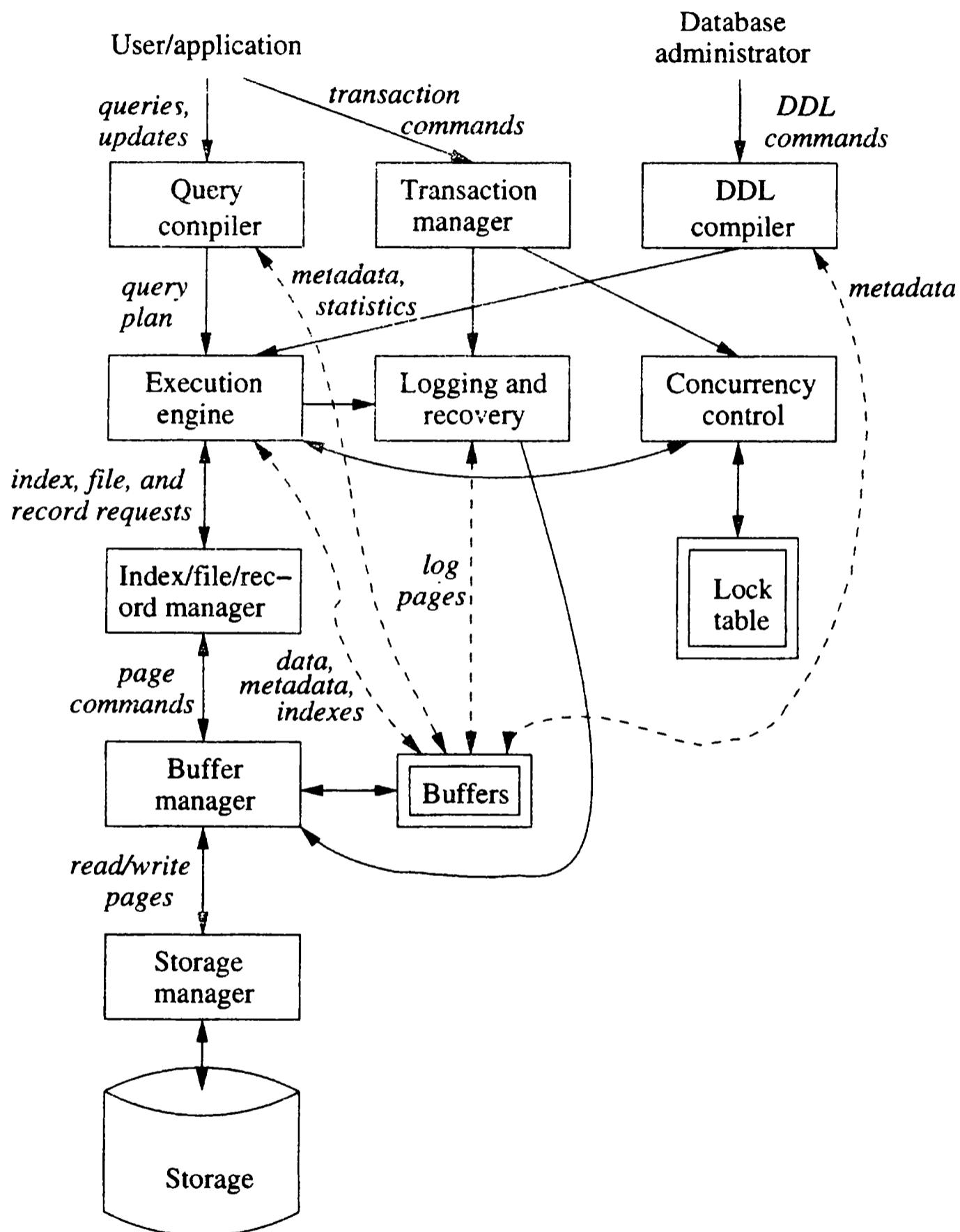


Figure 1.1: Database management system components

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 (“rollback” 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*.<sup>1</sup>

---

<sup>1</sup>Databases 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:

- a) The attributes of each relation.
- b) The tuples of each relation.
- c) The components of one tuple from each relation.
- d) The relation schema for each relation.
- e) The database schema.
- f) A suitable domain for each attribute.
- g) 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`',<sup>2</sup> 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).

---

<sup>2</sup>Notice 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.<sup>3</sup> 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

---

<sup>3</sup>The 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 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:

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

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>inColor</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>inColor</i>	<i>studioName</i>	<i>producerC#</i>
Star Wars	1977	124	true	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

<i>A</i>	<i>B</i>
1	2
3	4

(a) Relation *R*

<i>B</i>	<i>C</i>	<i>D</i>
2	5	6
4	7	8
9	10	11

(b) Relation *S*

<i>A</i>	<i>R.B</i>	<i>S.B</i>	<i>C</i>	<i>D</i>
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

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
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  $S$ ).

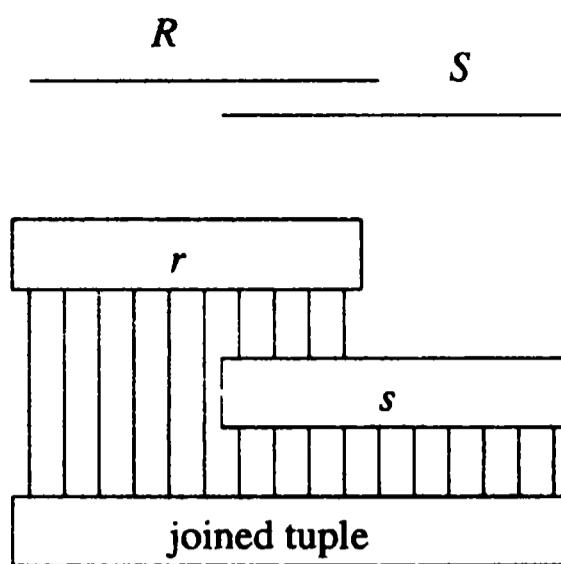


Figure 2.15: Joining tuples

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  $\theta$ .

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

<i>A</i>	<i>U.B</i>	<i>U.C</i>	<i>V.B</i>	<i>V.C</i>	<i>D</i>
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

<i>A</i>	<i>U.B</i>	<i>U.C</i>	<i>V.B</i>	<i>V.C</i>	<i>D</i>
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**.

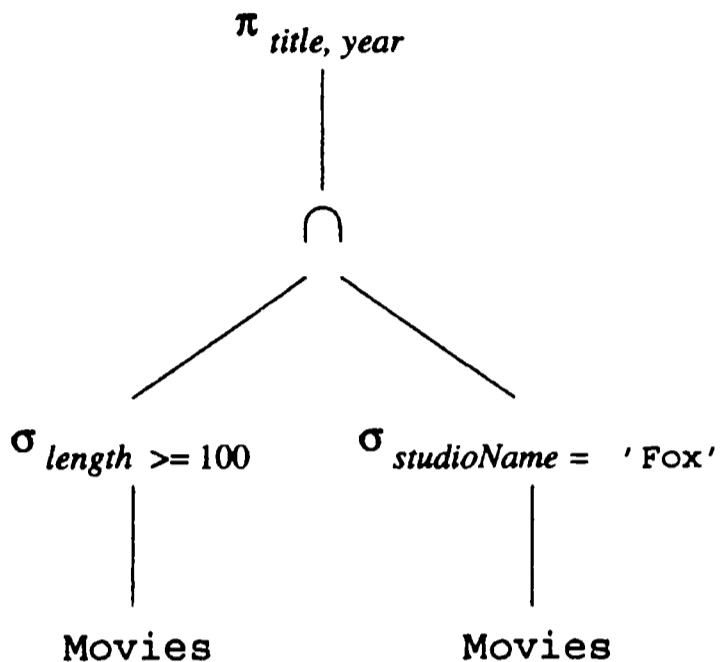


Figure 2.18: Expression tree for a relational algebra expression

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

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

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,

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

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.

□

### 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

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

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) := σlength ≥ 100(Movies)
S(t,y,l,i,s,p) := σstudioName = 'Fox'(Movies)
T(t,y,l,i,s,p) := R ∩ S
Answer(title, year) := πt,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) :=  $\sigma_{length \geq 100}(\text{Movies})$ 
S(t,y,l,i,s,p) :=  $\sigma_{studioName = 'Fox'}(\text{Movies})$ 
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.<sup>4</sup> 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.)

---

<sup>4</sup>Source: 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  $R \bowtie S$ , 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  $R \bowtie S$ .

**! Exercise 2.4.9:** The *antisemijoin*  $R \overline{\bowtie} S$  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  $R \overline{\bowtie} S$ .

**!! 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

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

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

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

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:

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

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:

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

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:

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

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

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

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/>



# Chapter 3

# Design Theory for Relational Databases

There are many ways we could go about designing a relational database schema for an application. In Chapter 4 we shall see several high-level notations for describing the structure of data and the ways in which these high-level designs can be converted into relations. We can also examine the requirements for a database and define relations directly, without going through a high-level intermediate stage. Whatever approach we use, it is common for an initial relational schema to have room for improvement, especially by eliminating redundancy. Often, the problems with a schema involve trying to combine too much into one relation.

Fortunately, there is a well developed theory for relational databases: “dependencies,” their implications for what makes a good relational database schema, and what we can do about a schema if it has flaws. In this chapter, we first identify the problems that are caused in some relation schemas by the presence of certain dependencies; these problems are referred to as “anomalies.”

Our discussion starts with “functional dependencies,” a generalization of the idea of a key for a relation. We then use the notion of functional dependencies to define normal forms for relation schemas. The impact of this theory, called “normalization,” is that we decompose relations into two or more relations when that will remove anomalies. Next, we introduce “multivalued dependencies,” which intuitively represent a condition where one or more attributes of a relation are independent from one or more other attributes. These dependencies also lead to normal forms and decomposition of relations to eliminate redundancy.

## 3.1 Functional Dependencies

There is a design theory for relations that lets us examine a design carefully and make improvements based on a few simple principles. The theory begins by

having us state the constraints that apply to the relation. The most common constraint is the “functional dependency,” a statement of a type that generalizes the idea of a key for a relation, which we introduced in Section 2.5.3. Later in this chapter, we shall see how this theory gives us simple tools to improve our designs by the process of “decomposition” of relations: the replacement of one relation by several, whose sets of attributes together include all the attributes of the original.

### 3.1.1 Definition of Functional Dependency

A *functional dependency* (FD) on a relation  $R$  is a statement of the form “If two tuples of  $R$  agree on all of the attributes  $A_1, A_2, \dots, A_n$  (i.e., the tuples have the same values in their respective components for each of these attributes), then they must also agree on all of another list of attributes  $B_1, B_2, \dots, B_m$ . We write this FD formally as  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  and say that

“ $A_1, A_2, \dots, A_n$  functionally determine  $B_1, B_2, \dots, B_m$ ”

Figure 3.1 suggests what this FD tells us about any two tuples  $t$  and  $u$  in the relation  $R$ . However, the  $A$ 's and  $B$ 's can be anywhere; it is not necessary for the  $A$ 's and  $B$ 's to appear consecutively or for the  $A$ 's to precede the  $B$ 's.

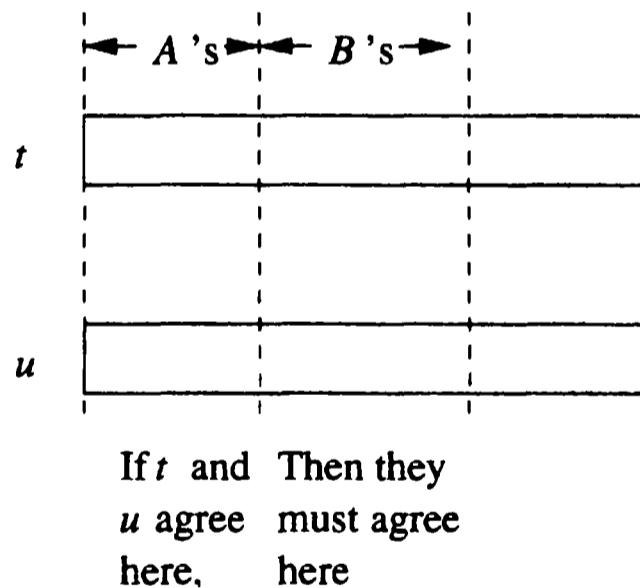


Figure 3.1: The effect of a functional dependency on two tuples.

If we can be sure every instance of a relation  $R$  will be one in which a given FD is true, then we say that  $R$  *satisfies* the FD. It is important to remember that when we say that  $R$  satisfies an FD  $f$ , we are asserting a constraint on  $R$ , not just saying something about one particular instance of  $R$ .

It is common for the right side of an FD to be a single attribute. In fact, we shall see that the one functional dependency  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  is equivalent to the set of FD's:

$$\begin{aligned} A_1 A_2 \cdots A_n &\rightarrow B_1 \\ A_1 A_2 \cdots A_n &\rightarrow B_2 \\ &\dots \\ A_1 A_2 \cdots A_n &\rightarrow B_m \end{aligned}$$

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>starName</i>
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 3.2: An instance of the relation `Movies1(title, year, length, genre, studioName, starName)`

**Example 3.1:** Let us consider the relation

`Movies1(title, year, length, genre, studioName, starName)`

an instance of which is shown in Fig. 3.2. While related to our running `Movies` relation, it has additional attributes, which is why we call it “`Movies1`” instead of “`Movies`.“ Notice that this relation tries to “do too much.” It holds information that in our running database schema was attributed to three different relations: `Movies`, `Studio`, and `StarsIn`. As we shall see, the schema for `Movies1` is not a good design. But to see what is wrong with the design, we must first determine the functional dependencies that hold for the relation. We claim that the following FD holds:

$$\text{title } \text{year} \rightarrow \text{length } \text{genre } \text{studioName}$$

Informally, this FD says that if two tuples have the same value in their `title` components, and they also have the same value in their `year` components, then these two tuples must also have the same values in their `length` components, the same values in their `genre` components, and the same values in their `studioName` components. This assertion makes sense, since we believe that it is not possible for there to be two movies released in the same year with the same title (although there could be movies of the same title released in different years). This point was discussed in Example 2.1. Thus, we expect that given a title and year, there is a unique movie. Therefore, there is a unique length for the movie, a unique genre, and a unique studio.

On the other hand, we observe that the statement

$$\text{title } \text{year} \rightarrow \text{starName}$$

is false; it is not a functional dependency. Given a movie, it is entirely possible that there is more than one star for the movie listed in our database. Notice that even had we been lazy and only listed one star for *Star Wars* and one star for *Wayne's World* (just as we only listed one of the many stars for *Gone With the Wind*), this FD would not suddenly become true for the relation `Movies1`.

The reason is that the FD says something about all possible instances of the relation, not about one of its instances. The fact that we *could* have an instance with multiple stars for a movie rules out the possibility that title and year functionally determine starName.  $\square$

### 3.1.2 Keys of Relations

We say a set of one or more attributes  $\{A_1, A_2, \dots, A_n\}$  is a *key* for a relation  $R$  if:

1. Those attributes functionally determine all other attributes of the relation. That is, it is impossible for two distinct tuples of  $R$  to agree on all of  $A_1, A_2, \dots, A_n$ .
2. No proper subset of  $\{A_1, A_2, \dots, A_n\}$  functionally determines all other attributes of  $R$ ; i.e., a key must be *minimal*.

When a key consists of a single attribute  $A$ , we often say that  $A$  (rather than  $\{A\}$ ) is a key.

**Example 3.2:** Attributes `{title, year, starName}` form a key for the relation `Movies1` of Fig. 3.2. First, we must show that they functionally determine all the other attributes. That is, suppose two tuples agree on these three attributes: `title`, `year`, and `starName`. Because they agree on `title` and `year`, they must agree on the other attributes — `length`, `genre`, and `studioName` — as we discussed in Example 3.1. Thus, two different tuples cannot agree on all of `title`, `year`, and `starName`; they would in fact be the same tuple.

Now, we must argue that no proper subset of `{title, year, starName}` functionally determines all other attributes. To see why, begin by observing that `title` and `year` do not determine `starName`, because many movies have more than one star. Thus, `{title, year}` is not a key.

`{year, starName}` is not a key because we could have a star in two movies in the same year; therefore

$$\text{year starName} \rightarrow \text{title}$$

is not an FD. Also, we claim that `{title, starName}` is not a key, because two movies with the same title, made in different years, occasionally have a star in common.<sup>1</sup>  $\square$

Sometimes a relation has more than one key. If so, it is common to designate one of the keys as the *primary key*. In commercial database systems, the choice of primary key can influence some implementation issues such as how the relation is stored on disk. However, the theory of FD's gives no special role to "primary keys."

---

<sup>1</sup>Since we asserted in an earlier book that there were no known examples of this phenomenon, several people have shown us we were wrong. It's an interesting challenge to discover stars that appeared in two versions of the same movie.

## What Is “Functional” About Functional Dependencies?

$A_1 A_2 \cdots A_n \rightarrow B$  is called a “functional” dependency because in principle there is a function that takes a list of values, one for each of attributes  $A_1, A_2, \dots, A_n$  and produces a unique value (or no value at all) for  $B$ . For instance, in the `Movies1` relation, we can imagine a function that takes a string like "Star Wars" and an integer like 1977 and produces the unique value of `length`, namely 124, that appears in the relation `Movies1`. However, this function is not the usual sort of function that we meet in mathematics, because there is no way to compute it from first principles. That is, we cannot perform some operations on strings like "Star Wars" and integers like 1977 and come up with the correct length. Rather, the function is only computed by lookup in the relation. We look for a tuple with the given `title` and `year` values and see what value that tuple has for `length`.

### 3.1.3 Superkeys

A set of attributes that contains a key is called a *superkey*, short for “superset of a key.” Thus, every key is a superkey. However, some superkeys are not (minimal) keys. Note that every superkey satisfies the first condition of a key: it functionally determines all other attributes of the relation. However, a superkey need not satisfy the second condition: minimality.

**Example 3.3:** In the relation of Example 3.2, there are many superkeys. Not only is the key

`{title, year, starName}`

a superkey, but any superset of this set of attributes, such as

`{title, year, starName, length, studioName}`

is a superkey.  $\square$

### 3.1.4 Exercises for Section 3.1

**Exercise 3.1.1:** Consider a relation about people in the United States, including their name, Social Security number, street address, city, state, ZIP code, area code, and phone number (7 digits). What FD’s would you expect to hold? What are the keys for the relation? To answer this question, you need to know something about the way these numbers are assigned. For instance, can an area

## Other Key Terminology

In some books and articles one finds different terminology regarding keys. One can find the term “key” used the way we have used the term “superkey,” that is, a set of attributes that functionally determine all the attributes, with no requirement of minimality. These sources typically use the term “candidate key” for a key that is minimal — that is, a “key” in the sense we use the term.

code straddle two states? Can a ZIP code straddle two area codes? Can two people have the same Social Security number? Can they have the same address or phone number?

**Exercise 3.1.2:** Consider a relation representing the present position of molecules in a closed container. The attributes are an ID for the molecule, the  $x$ ,  $y$ , and  $z$  coordinates of the molecule, and its velocity in the  $x$ ,  $y$ , and  $z$  dimensions. What FD’s would you expect to hold? What are the keys?

!! **Exercise 3.1.3:** Suppose  $R$  is a relation with attributes  $A_1, A_2, \dots, A_n$ . As a function of  $n$ , tell how many superkeys  $R$  has, if:

- a) The only key is  $A_1$ .
- b) The only keys are  $A_1$  and  $A_2$ .
- c) The only keys are  $\{A_1, A_2\}$  and  $\{A_3, A_4\}$ .
- d) The only keys are  $\{A_1, A_2\}$  and  $\{A_1, A_3\}$ .

## 3.2 Rules About Functional Dependencies

In this section, we shall learn how to *reason* about FD’s. That is, suppose we are told of a set of FD’s that a relation satisfies. Often, we can deduce that the relation must satisfy certain other FD’s. This ability to discover additional FD’s is essential when we discuss the design of good relation schemas in Section 3.3.

### 3.2.1 Reasoning About Functional Dependencies

Let us begin with a motivating example that will show us how we can infer a functional dependency from other given FD’s.

**Example 3.4:** If we are told that a relation  $R(A, B, C)$  satisfies the FD’s  $A \rightarrow B$  and  $B \rightarrow C$ , then we can deduce that  $R$  also satisfies the FD  $A \rightarrow C$ . How does that reasoning go? To prove that  $A \rightarrow C$ , we must consider two tuples of  $R$  that agree on  $A$  and prove they also agree on  $C$ .

Let the tuples agreeing on attribute  $A$  be  $(a, b_1, c_1)$  and  $(a, b_2, c_2)$ . Since  $R$  satisfies  $A \rightarrow B$ , and these tuples agree on  $A$ , they must also agree on  $B$ . That is,  $b_1 = b_2$ , and the tuples are really  $(a, b, c_1)$  and  $(a, b, c_2)$ , where  $b$  is both  $b_1$  and  $b_2$ . Similarly, since  $R$  satisfies  $B \rightarrow C$ , and the tuples agree on  $B$ , they agree on  $C$ . Thus,  $c_1 = c_2$ ; i.e., the tuples *do* agree on  $C$ . We have proved that any two tuples of  $R$  that agree on  $A$  also agree on  $C$ , and that is the FD  $A \rightarrow C$ .  $\square$

FD's often can be presented in several different ways, without changing the set of legal instances of the relation. We say:

- Two sets of FD's  $S$  and  $T$  are *equivalent* if the set of relation instances satisfying  $S$  is exactly the same as the set of relation instances satisfying  $T$ .
- More generally, a set of FD's  $S$  *follows* from a set of FD's  $T$  if every relation instance that satisfies all the FD's in  $T$  also satisfies all the FD's in  $S$ .

Note then that two sets of FD's  $S$  and  $T$  are equivalent if and only if  $S$  follows from  $T$ , and  $T$  follows from  $S$ .

In this section we shall see several useful rules about FD's. In general, these rules let us replace one set of FD's by an equivalent set, or to add to a set of FD's others that follow from the original set. An example is the *transitive rule* that lets us follow chains of FD's, as in Example 3.4. We shall also give an algorithm for answering the general question of whether one FD follows from one or more other FD's.

### 3.2.2 The Splitting/Combining Rule

Recall that in Section 3.1.1 we commented that the FD:

$$A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$$

was equivalent to the set of FD's:

$$A_1 A_2 \cdots A_n \rightarrow B_1, \quad A_1 A_2 \cdots A_n \rightarrow B_2, \dots, A_1 A_2 \cdots A_n \rightarrow B_m$$

That is, we may split attributes on the right side so that only one attribute appears on the right of each FD. Likewise, we can replace a collection of FD's having a common left side by a single FD with the same left side and all the right sides combined into one set of attributes. In either event, the new set of FD's is equivalent to the old. The equivalence noted above can be used in two ways.

- We can replace an FD  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  by a set of FD's  $A_1 A_2 \cdots A_n \rightarrow B_i$  for  $i = 1, 2, \dots, m$ . This transformation we call the *splitting rule*.

- We can replace a set of FD's  $A_1 A_2 \cdots A_n \rightarrow B_i$  for  $i = 1, 2, \dots, m$  by the single FD  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$ . We call this transformation the *combining rule*.

**Example 3.5:** In Example 3.1 the set of FD's:

$$\begin{aligned} & \text{title year} \rightarrow \text{length} \\ & \text{title year} \rightarrow \text{genre} \\ & \text{title year} \rightarrow \text{studioName} \end{aligned}$$

is equivalent to the single FD:

$$\text{title year} \rightarrow \text{length genre studioName}$$

that we asserted there.  $\square$

The reason the splitting and combining rules are true should be obvious. Suppose we have two tuples that agree in  $A_1, A_2, \dots, A_n$ . As a single FD, we would assert “then the tuples must agree in all of  $B_1, B_2, \dots, B_m$ .” As individual FD's, we assert “then the tuples agree in  $B_1$ , and they agree in  $B_2$ , and, . . . , and they agree in  $B_m$ .” These two conclusions say exactly the same thing.

One might imagine that splitting could be applied to the left sides of FD's as well as to right sides. However, there is no splitting rule for left sides, as the following example shows.

**Example 3.6:** Consider one of the FD's such as:

$$\text{title year} \rightarrow \text{length}$$

for the relation `Movies1` in Example 3.1. If we try to split the left side into

$$\begin{aligned} & \text{title} \rightarrow \text{length} \\ & \text{year} \rightarrow \text{length} \end{aligned}$$

then we get two false FD's. That is, `title` does not functionally determine `length`, since there can be several movies with the same title (e.g., *King Kong*) but of different lengths. Similarly, `year` does not functionally determine `length`, because there are certainly movies of different lengths made in any one year.  $\square$

### 3.2.3 Trivial Functional Dependencies

A constraint of any kind on a relation is said to be *trivial* if it holds for every instance of the relation, regardless of what other constraints are assumed. When the constraints are FD's, it is easy to tell whether an FD is trivial. They are the FD's  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  such that

$$\{B_1, B_2, \dots, B_m\} \subseteq \{A_1, A_2, \dots, A_n\}$$

That is, a trivial FD has a right side that is a subset of its left side. For example,

$$\text{title year} \rightarrow \text{title}$$

is a trivial FD, as is

$$\text{title} \rightarrow \text{title}$$

Every trivial FD holds in every relation, since it says that “two tuples that agree in all of  $A_1, A_2, \dots, A_n$  agree in a subset of them.” Thus, we may assume any trivial FD, without having to justify it on the basis of what FD’s are asserted for the relation.

There is an intermediate situation in which some, but not all, of the attributes on the right side of an FD are also on the left. This FD is not trivial, but it can be simplified by removing from the right side of an FD those attributes that appear on the left. That is:

- The FD  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  is equivalent to

$$A_1 A_2 \cdots A_n \rightarrow C_1 C_2 \cdots C_k$$

where the  $C$ ’s are all those  $B$ ’s that are not also  $A$ ’s.

We call this rule, illustrated in Fig. 3.3, the *trivial-dependency rule*.

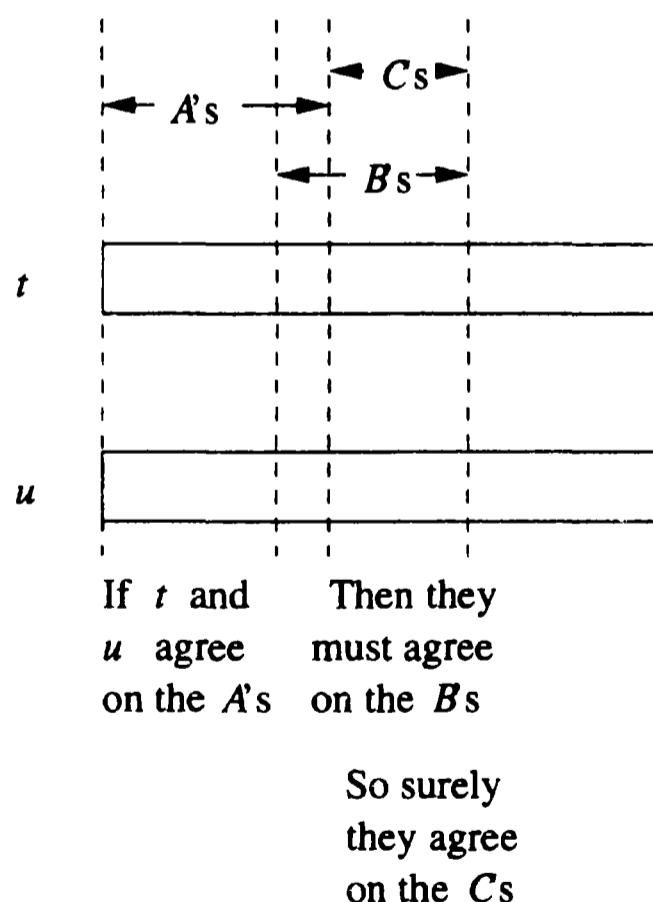


Figure 3.3: The trivial-dependency rule

### 3.2.4 Computing the Closure of Attributes

Before proceeding to other rules, we shall give a general principle from which all true rules follow. Suppose  $\{A_1, A_2, \dots, A_n\}$  is a set of attributes and  $S$

is a set of FD's. The *closure* of  $\{A_1, A_2, \dots, A_n\}$  under the FD's in  $S$  is the set of attributes  $B$  such that every relation that satisfies all the FD's in set  $S$  also satisfies  $A_1 A_2 \cdots A_n \rightarrow B$ . That is,  $A_1 A_2 \cdots A_n \rightarrow B$  follows from the FD's of  $S$ . We denote the closure of a set of attributes  $A_1 A_2 \cdots A_n$  by  $\{A_1, A_2, \dots, A_n\}^+$ . Note that  $A_1, A_2, \dots, A_n$  are always in  $\{A_1, A_2, \dots, A_n\}^+$  because the FD  $A_1 A_2 \cdots A_n \rightarrow A_i$  is trivial when  $i$  is one of  $1, 2, \dots, n$ .

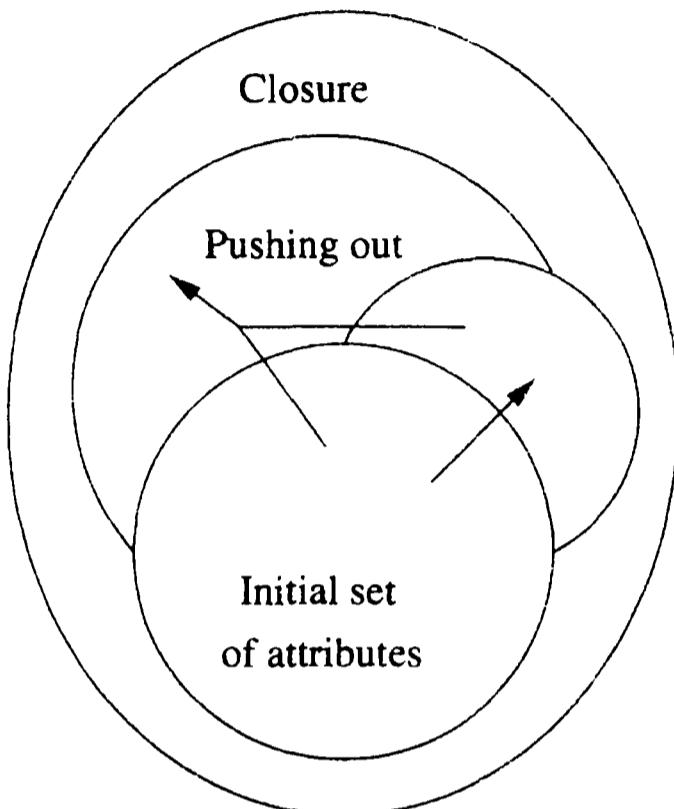


Figure 3.4: Computing the closure of a set of attributes

Figure 3.4 illustrates the closure process. Starting with the given set of attributes, we repeatedly expand the set by adding the right sides of FD's as soon as we have included their left sides. Eventually, we cannot expand the set any further, and the resulting set is the closure. More precisely:

**Algorithm 3.7:** Closure of a Set of Attributes.

**INPUT:** A set of attributes  $\{A_1, A_2, \dots, A_n\}$  and a set of FD's  $S$ .

**OUTPUT:** The closure  $\{A_1, A_2, \dots, A_n\}^+$ .

1. If necessary, split the FD's of  $S$ , so each FD in  $S$  has a single attribute on the right.
2. Let  $X$  be a set of attributes that eventually will become the closure. Initialize  $X$  to be  $\{A_1, A_2, \dots, A_n\}$ .
3. Repeatedly search for some FD

$$B_1 B_2 \cdots B_m \rightarrow C$$

such that all of  $B_1, B_2, \dots, B_m$  are in the set of attributes  $X$ , but  $C$  is not. Add  $C$  to the set  $X$  and repeat the search. Since  $X$  can only grow, and the number of attributes of any relation schema must be finite, eventually nothing more can be added to  $X$ , and this step ends.

4. The set  $X$ , after no more attributes can be added to it, is the correct value of  $\{A_1, A_2, \dots, A_n\}^+$ .

□

**Example 3.8:** Let us consider a relation with attributes  $A, B, C, D, E$ , and  $F$ . Suppose that this relation has the FD's  $AB \rightarrow C$ ,  $BC \rightarrow AD$ ,  $D \rightarrow E$ , and  $CF \rightarrow B$ . What is the closure of  $\{A, B\}$ , that is,  $\{A, B\}^+$ ?

First, split  $BC \rightarrow AD$  into  $BC \rightarrow A$  and  $BC \rightarrow D$ . Then, start with  $X = \{A, B\}$ . First, notice that both attributes on the left side of FD  $AB \rightarrow C$  are in  $X$ , so we may add the attribute  $C$ , which is on the right side of that FD. Thus, after one iteration of Step 3,  $X$  becomes  $\{A, B, C\}$ .

Next, we see that the left sides of  $BC \rightarrow A$  and  $BC \rightarrow D$  are now contained in  $X$ , so we may add to  $X$  the attributes  $A$  and  $D$ .  $A$  is already there, but  $D$  is not, so  $X$  next becomes  $\{A, B, C, D\}$ . At this point, we may use the FD  $D \rightarrow E$  to add  $E$  to  $X$ , which is now  $\{A, B, C, D, E\}$ . No more changes to  $X$  are possible. In particular, the FD  $CF \rightarrow B$  can not be used, because its left side never becomes contained in  $X$ . Thus,  $\{A, B\}^+ = \{A, B, C, D, E\}$ . □

By computing the closure of any set of attributes, we can test whether any given FD  $A_1 A_2 \cdots A_n \rightarrow B$  follows from a set of FD's  $S$ . First compute  $\{A_1, A_2, \dots, A_n\}^+$  using the set of FD's  $S$ . If  $B$  is in  $\{A_1, A_2, \dots, A_n\}^+$ , then  $A_1 A_2 \cdots A_n \rightarrow B$  does follow from  $S$ , and if  $B$  is not in  $\{A_1, A_2, \dots, A_n\}^+$ , then this FD does not follow from  $S$ . More generally,  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  follows from set of FD's  $S$  if and only if all of  $B_1, B_2, \dots, B_m$  are in

$$\{A_1, A_2, \dots, A_n\}^+$$

**Example 3.9:** Consider the relation and FD's of Example 3.8. Suppose we wish to test whether  $AB \rightarrow D$  follows from these FD's. We compute  $\{A, B\}^+$ , which is  $\{A, B, C, D, E\}$ , as we saw in that example. Since  $D$  is a member of the closure, we conclude that  $AB \rightarrow D$  does follow.

On the other hand, consider the FD  $D \rightarrow A$ . To test whether this FD follows from the given FD's, first compute  $\{D\}^+$ . To do so, we start with  $X = \{D\}$ . We can use the FD  $D \rightarrow E$  to add  $E$  to the set  $X$ . However, then we are stuck. We cannot find any other FD whose left side is contained in  $X = \{D, E\}$ , so  $\{D\}^+ = \{D, E\}$ . Since  $A$  is not a member of  $\{D, E\}$ , we conclude that  $D \rightarrow A$  does not follow. □

### 3.2.5 Why the Closure Algorithm Works

In this section, we shall show why Algorithm 3.7 correctly decides whether or not an FD  $A_1 A_2 \cdots A_n \rightarrow B$  follows from a given set of FD's  $S$ . There are two parts to the proof:

1. We must prove that Algorithm 3.7 does not claim too much. That is, we must show that if  $A_1 A_2 \cdots A_n \rightarrow B$  is asserted by the closure test (i.e.,

$B$  is in  $\{A_1, A_2, \dots, A_n\}^+$ , then  $A_1 A_2 \cdots A_n \rightarrow B$  holds in any relation that satisfies all the FD's in  $S$ .

2. We must prove that Algorithm 3.7 does not fail to discover a FD that truly follows from the set of FD's  $S$ .

### Why the Closure Algorithm Claims only True FD's

We can prove by induction on the number of times that we apply the growing operation of Step 3 that for every attribute  $D$  in  $X$ , the FD  $A_1 A_2 \cdots A_n \rightarrow D$  holds. That is, every relation  $R$  satisfying all of the FD's in  $S$  also satisfies  $A_1 A_2 \cdots A_n \rightarrow D$ .

**BASIS:** The basis case is when there are zero steps. Then  $D$  must be one of  $A_1, A_2, \dots, A_n$ , and surely  $A_1 A_2 \cdots A_n \rightarrow D$  holds in any relation, because it is a trivial FD.

**INDUCTION:** For the induction, suppose  $D$  was added when we used the FD  $B_1 B_2 \cdots B_m \rightarrow D$  of  $S$ . We know by the inductive hypothesis that  $R$  satisfies  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$ . Now, suppose two tuples of  $R$  agree on all of  $A_1, A_2, \dots, A_n$ . Then since  $R$  satisfies  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$ , the two tuples must agree on all of  $B_1, B_2, \dots, B_m$ . Since  $R$  satisfies  $B_1 B_2 \cdots B_m \rightarrow D$ , we also know these two tuples agree on  $D$ . Thus,  $R$  satisfies  $A_1 A_2 \cdots A_n \rightarrow D$ .

### Why the Closure Algorithm Discovers All True FD's

Suppose  $A_1 A_2 \cdots A_n \rightarrow B$  were an FD that Algorithm 3.7 says does not follow from set  $S$ . That is, the closure of  $\{A_1, A_2, \dots, A_n\}$  using set of FD's  $S$  does not include  $B$ . We must show that FD  $A_1 A_2 \cdots A_n \rightarrow B$  really doesn't follow from  $S$ . That is, we must show that there is at least one relation instance that satisfies all the FD's in  $S$ , and yet does not satisfy  $A_1 A_2 \cdots A_n \rightarrow B$ .

This instance  $I$  is actually quite simple to construct; it is shown in Fig. 3.5.  $I$  has only two tuples:  $t$  and  $s$ . The two tuples agree in all the attributes of  $\{A_1, A_2, \dots, A_n\}^+$ , and they disagree in all the other attributes. We must show first that  $I$  satisfies all the FD's of  $S$ , and then that it does not satisfy  $A_1 A_2 \cdots A_n \rightarrow B$ .

	$\{A_1, A_2, \dots, A_n\}^+$	Other Attributes
$t:$	1 1 1 … 1 1	0 0 0 … 0 0
$s:$	1 1 1 … 1 1	1 1 1 … 1 1

Figure 3.5: An instance  $I$  satisfying  $S$  but not  $A_1 A_2 \cdots A_n \rightarrow B$

Suppose there were some FD  $C_1 C_2 \cdots C_k \rightarrow D$  in set  $S$  (after splitting right sides) that instance  $I$  does not satisfy. Since  $I$  has only two tuples,  $t$  and  $s$ , those must be the two tuples that violate  $C_1 C_2 \cdots C_k \rightarrow D$ . That is,  $t$  and  $s$  agree in all the attributes of  $\{C_1, C_2, \dots, C_k\}$ , yet disagree on  $D$ . If we

examine Fig. 3.5 we see that all of  $C_1, C_2, \dots, C_k$  must be among the attributes of  $\{A_1, A_2, \dots, A_n\}^+$ , because those are the only attributes on which  $t$  and  $s$  agree. Likewise,  $D$  must be among the other attributes, because only on those attributes do  $t$  and  $s$  disagree.

But then we did not compute the closure correctly.  $C_1C_2 \cdots C_k \rightarrow D$  should have been applied when  $X$  was  $\{A_1, A_2, \dots, A_n\}$  to add  $D$  to  $X$ . We conclude that  $C_1C_2 \cdots C_k \rightarrow D$  cannot exist; i.e., instance  $I$  satisfies  $S$ .

Second, we must show that  $I$  does not satisfy  $A_1A_2 \cdots A_n \rightarrow B$ . However, this part is easy. Surely,  $A_1, A_2, \dots, A_n$  are among the attributes on which  $t$  and  $s$  agree. Also, we know that  $B$  is not in  $\{A_1, A_2, \dots, A_n\}^+$ , so  $B$  is one of the attributes on which  $t$  and  $s$  disagree. Thus,  $I$  does not satisfy  $A_1A_2 \cdots A_n \rightarrow B$ . We conclude that Algorithm 3.7 asserts neither too few nor too many FD's; it asserts exactly those FD's that do follow from  $S$ .

### 3.2.6 The Transitive Rule

The transitive rule lets us cascade two FD's, and generalizes the observation of Example 3.4.

- If  $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$  and  $B_1B_2 \cdots B_m \rightarrow C_1C_2 \cdots C_k$  hold in relation  $R$ , then  $A_1A_2 \cdots A_n \rightarrow C_1C_2 \cdots C_k$  also holds in  $R$ .

If some of the  $C$ 's are among the  $A$ 's, we may eliminate them from the right side by the trivial-dependencies rule.

To see why the transitive rule holds, apply the test of Section 3.2.4. To test whether  $A_1A_2 \cdots A_n \rightarrow C_1C_2 \cdots C_k$  holds, we need to compute the closure  $\{A_1, A_2, \dots, A_n\}^+$  with respect to the two given FD's.

The FD  $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$  tells us that all of  $B_1, B_2, \dots, B_m$  are in  $\{A_1, A_2, \dots, A_n\}^+$ . Then, we can use the FD  $B_1B_2 \cdots B_m \rightarrow C_1C_2 \cdots C_k$  to add  $C_1, C_2, \dots, C_k$  to  $\{A_1, A_2, \dots, A_n\}^+$ . Since all the  $C$ 's are in

$$\{A_1, A_2, \dots, A_n\}^+$$

we conclude that  $A_1A_2 \cdots A_n \rightarrow C_1C_2 \cdots C_k$  holds for any relation that satisfies both  $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$  and  $B_1B_2 \cdots B_m \rightarrow C_1C_2 \cdots C_k$ .

**Example 3.10:** Here is another version of the **Movies** relation that includes both the studio of the movie and some information about that studio.

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>studioAddr</i>
Star Wars	1977	124	sciFi	Fox	Hollywood
Eight Below	2005	120	drama	Disney	Buena Vista
Wayne's World	1992	95	comedy	Paramount	Hollywood

Two of the FD's that we might reasonably claim to hold are:

$$\begin{aligned} \textit{title year} &\rightarrow \textit{studioName} \\ \textit{studioName} &\rightarrow \textit{studioAddr} \end{aligned}$$

## Closures and Keys

Notice that  $\{A_1, A_2, \dots, A_n\}^+$  is the set of all attributes of a relation if and only if  $A_1, A_2, \dots, A_n$  is a superkey for the relation. For only then does  $A_1, A_2, \dots, A_n$  functionally determine all the other attributes. We can test if  $A_1, A_2, \dots, A_n$  is a key for a relation by checking first that  $\{A_1, A_2, \dots, A_n\}^+$  is all attributes, and then checking that, for no set  $X$  formed by removing one attribute from  $\{A_1, A_2, \dots, A_n\}$ , is  $X^+$  the set of all attributes.

The first is justified because there can be only one movie with a given title and year, and there is only one studio that owns a given movie. The second is justified because studios have unique addresses.

The transitive rule allows us to combine the two FD's above to get a new FD:

$$\text{title year} \rightarrow \text{studioAddr}$$

This FD says that a title and year (i.e., a movie) determines an address — the address of the studio owning the movie.  $\square$

### 3.2.7 Closing Sets of Functional Dependencies

Sometimes we have a choice of which FD's we use to represent the full set of FD's for a relation. If we are given a set of FD's  $S$  (such as the FD's that hold in a given relation), then any set of FD's equivalent to  $S$  is said to be a *basis* for  $S$ . To avoid some of the explosion of possible bases, we shall limit ourselves to considering only bases whose FD's have singleton right sides. If we have any basis, we can apply the splitting rule to make the right sides be singletons. A *minimal basis* for a relation is a basis  $B$  that satisfies three conditions:

1. All the FD's in  $B$  have singleton right sides.
2. If any FD is removed from  $B$ , the result is no longer a basis.
3. If for any FD in  $B$  we remove one or more attributes from the left side of  $F$ , the result is no longer a basis.

Notice that no trivial FD can be in a minimal basis, because it could be removed by rule (2).

**Example 3.11:** Consider a relation  $R(A, B, C)$  such that each attribute functionally determines the other two attributes. The full set of derived FD's thus includes six FD's with one attribute on the left and one on the right;  $A \rightarrow B$ ,  $A \rightarrow C$ ,  $B \rightarrow A$ ,  $B \rightarrow C$ ,  $C \rightarrow A$ , and  $C \rightarrow B$ . It also includes the three

## A Complete Set of Inference Rules

If we want to know whether one FD follows from some given FD's, the closure computation of Section 3.2.4 will always serve. However, it is interesting to know that there is a set of rules, called *Armstrong's axioms*, from which it is possible to derive any FD that follows from a given set. These axioms are:

1. *Reflexivity.* If  $\{B_1, B_2, \dots, B_m\} \subseteq \{A_1, A_2, \dots, A_n\}$ , then  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$ . These are what we have called trivial FD's.
2. *Augmentation.* If  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$ , then

$$A_1 A_2 \cdots A_n C_1 C_2 \cdots C_k \rightarrow B_1 B_2 \cdots B_m C_1 C_2 \cdots C_k$$

for any set of attributes  $C_1, C_2, \dots, C_k$ . Since some of the  $C$ 's may also be  $A$ 's or  $B$ 's or both, we should eliminate from the left side duplicate attributes and do the same for the right side.

3. *Transitivity.* If

$$A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m \text{ and } B_1 B_2 \cdots B_m \rightarrow C_1 C_2 \cdots C_k$$

then  $A_1 A_2 \cdots A_n \rightarrow C_1 C_2 \cdots C_k$ .

nontrivial FD's with two attributes on the left:  $AB \rightarrow C$ ,  $AC \rightarrow B$ , and  $BC \rightarrow A$ . There are also FD's with more than one attribute on the right, such as  $A \rightarrow BC$ , and trivial FD's such as  $A \rightarrow A$ .

Relation  $R$  and its FD's have several minimal bases. One is

$$\{A \rightarrow B, B \rightarrow A, B \rightarrow C, C \rightarrow B\}$$

Another is  $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$ . There are several other minimal bases for  $R$ , and we leave their discovery as an exercise.  $\square$

### 3.2.8 Projecting Functional Dependencies

When we study design of relation schemas, we shall also have need to answer the following question about FD's. Suppose we have a relation  $R$  with set of FD's  $S$ , and we project  $R$  by computing  $R_1 = \pi_L(R)$ , for some list of attributes  $L$ . What FD's hold in  $R_1$ ?

The answer is obtained in principle by computing the *projection of functional dependencies*  $S'$ , which is all FD's that:

- a) Follow from  $S$ , and
- b) Involve only attributes of  $R_1$ .

Since there may be a large number of such FD's, and many of them may be redundant (i.e., they follow from other such FD's), we are free to simplify that set of FD's if we wish. However, in general, the calculation of the FD's for  $R_1$  is exponential in the number of attributes of  $R_1$ . The simple algorithm is summarized below.

**Algorithm 3.12:** Projecting a Set of Functional Dependencies.

**INPUT:** A relation  $R$  and a second relation  $R_1$  computed by the projection  $R_1 = \pi_L(R)$ . Also, a set of FD's  $S$  that hold in  $R$ .

**OUTPUT:** The set of FD's that hold in  $R_1$ .

**METHOD:**

1. Let  $T$  be the eventual output set of FD's. Initially,  $T$  is empty.
2. For each set of attributes  $X$  that is a subset of the attributes of  $R_1$ , compute  $X^+$ . This computation is performed with respect to the set of FD's  $S$ , and may involve attributes that are in the schema of  $R$  but not  $R_1$ . Add to  $T$  all nontrivial FD's  $X \rightarrow A$  such that  $A$  is both in  $X^+$  and an attribute of  $R_1$ .
3. Now,  $T$  is a basis for the FD's that hold in  $R_1$ , but may not be a minimal basis. We may construct a minimal basis by modifying  $T$  as follows:
  - (a) If there is an FD  $F$  in  $T$  that follows from the other FD's in  $T$ , remove  $F$  from  $T$ .
  - (b) Let  $Y \rightarrow B$  be an FD in  $T$ , with at least two attributes in  $Y$ , and let  $Z$  be  $Y$  with one of its attributes removed. If  $Z \rightarrow B$  follows from the FD's in  $T$  (including  $Y \rightarrow B$ ), then replace  $Y \rightarrow B$  by  $Z \rightarrow B$ .
  - (c) Repeat the above steps in all possible ways until no more changes to  $T$  can be made.

□

**Example 3.13:** Suppose  $R(A, B, C, D)$  has FD's  $A \rightarrow B$ ,  $B \rightarrow C$ , and  $C \rightarrow D$ . Suppose also that we wish to project out the attribute  $B$ , leaving a relation  $R_1(A, C, D)$ . In principle, to find the FD's for  $R_1$ , we need to take the closure of all eight subsets of  $\{A, C, D\}$ , using the full set of FD's, including those involving  $B$ . However, there are some obvious simplifications we can make.

- Closing the empty set and the set of all attributes cannot yield a nontrivial FD.

- If we already know that the closure of some set  $X$  is all attributes, then we cannot discover any new FD's by closing supersets of  $X$ .

Thus, we may start with the closures of the singleton sets, and then move on to the doubleton sets if necessary. For each closure of a set  $X$ , we add the FD  $X \rightarrow E$  for each attribute  $E$  that is in  $X^+$  and in the schema of  $R_1$ , but not in  $X$ .

First,  $\{A\}^+ = \{A, B, C, D\}$ . Thus,  $A \rightarrow C$  and  $A \rightarrow D$  hold in  $R_1$ . Note that  $A \rightarrow B$  is true in  $R$ , but makes no sense in  $R_1$  because  $B$  is not an attribute of  $R_1$ .

Next, we consider  $\{C\}^+ = \{C, D\}$ , from which we get the additional FD  $C \rightarrow D$  for  $R_1$ . Since  $\{D\}^+ = \{D\}$ , we can add no more FD's, and are done with the singletons.

Since  $\{A\}^+$  includes all attributes of  $R_1$ , there is no point in considering any superset of  $\{A\}$ . The reason is that whatever FD we could discover, for instance  $AC \rightarrow D$ , follows from an FD with only  $A$  on the left side:  $A \rightarrow D$  in this case. Thus, the only doubleton whose closure we need to take is  $\{C, D\}^+ = \{C, D\}$ . This observation allows us to add nothing. We are done with the closures, and the FD's we have discovered are  $A \rightarrow C$ ,  $A \rightarrow D$ , and  $C \rightarrow D$ .

If we wish, we can observe that  $A \rightarrow D$  follows from the other two by transitivity. Therefore a simpler, equivalent set of FD's for  $R_1$  is  $A \rightarrow C$  and  $C \rightarrow D$ . This set is, in fact, a minimal basis for the FD's of  $R_1$ .  $\square$

### 3.2.9 Exercises for Section 3.2

**Exercise 3.2.1:** Consider a relation with schema  $R(A, B, C, D)$  and FD's  $AB \rightarrow C$ ,  $C \rightarrow D$ , and  $D \rightarrow A$ .

- What are all the nontrivial FD's that follow from the given FD's? You should restrict yourself to FD's with single attributes on the right side.
- What are all the keys of  $R$ ?
- What are all the superkeys for  $R$  that are not keys?

**Exercise 3.2.2:** Repeat Exercise 3.2.1 for the following schemas and sets of FD's:

- $S(A, B, C, D)$  with FD's  $A \rightarrow B$ ,  $B \rightarrow C$ , and  $B \rightarrow D$ .
- $T(A, B, C, D)$  with FD's  $AB \rightarrow C$ ,  $BC \rightarrow D$ ,  $CD \rightarrow A$ , and  $AD \rightarrow B$ .
- $U(A, B, C, D)$  with FD's  $A \rightarrow B$ ,  $B \rightarrow C$ ,  $C \rightarrow D$ , and  $D \rightarrow A$ .

**Exercise 3.2.3:** Show that the following rules hold, by using the closure test of Section 3.2.4.

- Augmenting left sides.* If  $A_1 A_2 \cdots A_n \rightarrow B$  is an FD, and  $C$  is another attribute, then  $A_1 A_2 \cdots A_n C \rightarrow B$  follows.

- b) *Full augmentation.* If  $A_1A_2 \cdots A_n \rightarrow B$  is an FD, and  $C$  is another attribute, then  $A_1A_2 \cdots A_nC \rightarrow BC$  follows. Note: from this rule, the “augmentation” rule mentioned in the box of Section 3.2.7 on “A Complete Set of Inference Rules” can easily be proved.
- c) *Pseudotransitivity.* Suppose FD’s  $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$  and

$$C_1C_2 \cdots C_k \rightarrow D$$

hold, and the  $B$ ’s are each among the  $C$ ’s. Then

$$A_1A_2 \cdots A_nE_1E_2 \cdots E_j \rightarrow D$$

holds, where the  $E$ ’s are all those of the  $C$ ’s that are not found among the  $B$ ’s.

- d) *Addition.* If FD’s  $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$  and

$$C_1C_2 \cdots C_k \rightarrow D_1D_2 \cdots D_j$$

hold, then FD  $A_1A_2 \cdots A_nC_1C_2 \cdots C_k \rightarrow B_1B_2 \cdots B_mD_1D_2 \cdots D_j$  also holds. In the above, we should remove one copy of any attribute that appears among both the  $A$ ’s and  $C$ ’s or among both the  $B$ ’s and  $D$ ’s.

**! Exercise 3.2.4:** Show that each of the following are *not* valid rules about FD’s by giving example relations that satisfy the given FD’s (following the “if”) but not the FD that allegedly follows (after the “then”).

- a) If  $A \rightarrow B$  then  $B \rightarrow A$ .
- b) If  $AB \rightarrow C$  and  $A \rightarrow C$ , then  $B \rightarrow C$ .
- c) If  $AB \rightarrow C$ , then  $A \rightarrow C$  or  $B \rightarrow C$ .

**! Exercise 3.2.5:** Show that if a relation has no attribute that is functionally determined by all the other attributes, then the relation has no nontrivial FD’s at all.

**! Exercise 3.2.6:** Let  $X$  and  $Y$  be sets of attributes. Show that if  $X \subseteq Y$ , then  $X^+ \subseteq Y^+$ , where the closures are taken with respect to the same set of FD’s.

**! Exercise 3.2.7:** Prove that  $(X^+)^+ = X^+$ .

**!! Exercise 3.2.8:** We say a set of attributes  $X$  is *closed* (with respect to a given set of FD’s) if  $X^+ = X$ . Consider a relation with schema  $R(A, B, C, D)$  and an unknown set of FD’s. If we are told which sets of attributes are closed, we can discover the FD’s. What are the FD’s if:

- a) All sets of the four attributes are closed.

- b) The only closed sets are  $\emptyset$  and  $\{A, B, C, D\}$ .
- c) The closed sets are  $\emptyset$ ,  $\{A, B\}$ , and  $\{A, B, C, D\}$ .

**! Exercise 3.2.9:** Find all the minimal bases for the FD's and relation of Example 3.11.

**! Exercise 3.2.10:** Suppose we have relation  $R(A, B, C, D, E)$ , with some set of FD's, and we wish to project those FD's onto relation  $S(A, B, C)$ . Give the FD's that hold in  $S$  if the FD's for  $R$  are:

- a)  $AB \rightarrow DE$ ,  $C \rightarrow E$ ,  $D \rightarrow C$ , and  $E \rightarrow A$ .
- b)  $A \rightarrow D$ ,  $BD \rightarrow E$ ,  $AC \rightarrow E$ , and  $DE \rightarrow B$ .
- c)  $AB \rightarrow D$ ,  $AC \rightarrow E$ ,  $BC \rightarrow D$ ,  $D \rightarrow A$ , and  $E \rightarrow B$ .
- d)  $A \rightarrow B$ ,  $B \rightarrow C$ ,  $C \rightarrow D$ ,  $D \rightarrow E$ , and  $E \rightarrow A$ .

In each case, it is sufficient to give a minimal basis for the full set of FD's of  $S$ .

**!! Exercise 3.2.11:** Show that if an FD  $F$  follows from some given FD's, then we can prove  $F$  from the given FD's using Armstrong's axioms (defined in the box “A Complete Set of Inference Rules” in Section 3.2.7). *Hint:* Examine Algorithm 3.7 and show how each step of that algorithm can be mimicked by inferring some FD's by Armstrong's axioms.

### 3.3 Design of Relational Database Schemas

Careless selection of a relational database schema can lead to redundancy and related anomalies. For instance, consider the relation in Fig. 3.2, which we reproduce here as Fig. 3.6. Notice that the length and genre for *Star Wars* and *Wayne's World* are each repeated, once for each star of the movie. The repetition of this information is redundant. It also introduces the potential for several kinds of errors, as we shall see.

In this section, we shall tackle the problem of design of good relation schemas in the following stages:

1. We first explore in more detail the problems that arise when our schema is poorly designed.
2. Then, we introduce the idea of “decomposition,” breaking a relation schema (set of attributes) into two smaller schemas.
3. Next, we introduce “Boyce-Codd normal form,” or “BCNF,” a condition on a relation schema that eliminates these problems.
4. These points are tied together when we explain how to assure the BCNF condition by decomposing relation schemas.

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>starName</i>
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 3.6: The relation `Movies1` exhibiting anomalies

### 3.3.1 Anomalies

Problems such as redundancy that occur when we try to cram too much into a single relation are called *anomalies*. The principal kinds of anomalies that we encounter are:

1. *Redundancy.* Information may be repeated unnecessarily in several tuples. Examples are the length and genre for movies in Fig. 3.6.
2. *Update Anomalies.* We may change information in one tuple but leave the same information unchanged in another. For example, if we found that *Star Wars* is really 125 minutes long, we might carelessly change the length in the first tuple of Fig. 3.6 but not in the second or third tuples. You might argue that one should never be so careless, but it is possible to redesign relation `Movies1` so that the risk of such mistakes does not exist.
3. *Deletion Anomalies.* If a set of values becomes empty, we may lose other information as a side effect. For example, should we delete Vivien Leigh from the set of stars of *Gone With the Wind*, then we have no more stars for that movie in the database. The last tuple for *Gone With the Wind* in the relation `Movies1` would disappear, and with it information that it is 231 minutes long and a drama.

### 3.3.2 Decomposing Relations

The accepted way to eliminate these anomalies is to *decompose* relations. Decomposition of  $R$  involves splitting the attributes of  $R$  to make the schemas of two new relations. After describing the decomposition process, we shall show how to pick a decomposition that eliminates anomalies.

Given a relation  $R(A_1, A_2, \dots, A_n)$ , we may *decompose*  $R$  into two relations  $S(B_1, B_2, \dots, B_m)$  and  $T(C_1, C_2, \dots, C_k)$  such that:

1.  $\{A_1, A_2, \dots, A_n\} = \{B_1, B_2, \dots, B_m\} \cup \{C_1, C_2, \dots, C_k\}$ .
2.  $S = \pi_{B_1, B_2, \dots, B_m}(R)$ .

$$3. T = \pi_{C_1, C_2, \dots, C_k}(R).$$

**Example 3.14:** Let us decompose the **Movies1** relation of Fig. 3.6. Our choice, whose merit will be seen in Section 3.3.3, is to use:

1. A relation called **Movies2**, whose schema is all the attributes except for **starName**.
2. A relation called **Movies3**, whose schema consists of the attributes **title**, **year**, and **starName**.

The projection of **Movies1** onto these two new schemas is shown in Fig. 3.7.

□

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

(b) The relation **Movies2**.

<i>title</i>	<i>year</i>	<i>starName</i>
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Gone With the Wind	1939	Vivien Leigh
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

(b) The relation **Movies3**.

Figure 3.7: Projections of relation **Movies1**

Notice how this decomposition eliminates the anomalies we mentioned in Section 3.3.1. The redundancy has been eliminated; for example, the length of each film appears only once, in relation **Movies2**. The risk of an update anomaly is gone. For instance, since we only have to change the length of *Star Wars* in one tuple of **Movies2**, we cannot wind up with two different lengths for that movie.

Finally, the risk of a deletion anomaly is gone. If we delete all the stars for *Gone With the Wind*, say, that deletion makes the movie disappear from **Movies3**. But all the other information about the movie can still be found in **Movies2**.

It might appear that `Movies3` still has redundancy, since the title and year of a movie can appear several times. However, these two attributes form a key for movies, and there is no more succinct way to represent a movie. Moreover, `Movies3` does not offer an opportunity for an update anomaly. For instance, one might suppose that if we changed to 2008 the year in the Carrie Fisher tuple, but not the other two tuples for *Star Wars*, then there would be an update anomaly. However, there is nothing in our assumed FD's that prevents there being a different movie named *Star Wars* in 2008, and Carrie Fisher may star in that one as well. Thus, we do not want to prevent changing the year in one *Star Wars* tuple, nor is such a change necessarily incorrect.

### 3.3.3 Boyce-Codd Normal Form

The goal of decomposition is to replace a relation by several that do not exhibit anomalies. There is, it turns out, a simple condition under which the anomalies discussed above can be guaranteed not to exist. This condition is called *Boyce-Codd normal form*, or *BCNF*.

- A relation  $R$  is in BCNF if and only if: whenever there is a nontrivial FD  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  for  $R$ , it is the case that  $\{A_1, A_2, \dots, A_n\}$  is a superkey for  $R$ .

That is, the left side of every nontrivial FD must be a superkey. Recall that a superkey need not be minimal. Thus, an equivalent statement of the BCNF condition is that the left side of every nontrivial FD must contain a key.

**Example 3.15:** Relation `Movies1`, as in Fig. 3.6, is not in BCNF. To see why, we first need to determine what sets of attributes are keys. We argued in Example 3.2 why `{title, year, starName}` is a key. Thus, any set of attributes containing these three is a superkey. The same arguments we followed in Example 3.2 can be used to explain why no set of attributes that does not include all three of `title`, `year`, and `starName` could be a superkey. Thus, we assert that `{title, year, starName}` is the only key for `Movies1`.

However, consider the FD

$$\text{title year} \rightarrow \text{length genre studioName}$$

which holds in `Movies1` according to our discussion in Example 3.2.

Unfortunately, the left side of the above FD is not a superkey. In particular, we know that `title` and `year` do not functionally determine the sixth attribute, `starName`. Thus, the existence of this FD violates the BCNF condition and tells us `Movies1` is not in BCNF.  $\square$

**Example 3.16:** On the other hand, `Movies2` of Fig. 3.7 is in BCNF. Since

$$\text{title year} \rightarrow \text{length genre studioName}$$

holds in this relation, and we have argued that neither `title` nor `year` by itself functionally determines any of the other attributes, the only key for `Movies2` is  $\{\text{title}, \text{year}\}$ . Moreover, the only nontrivial FD's must have at least `title` and `year` on the left side, and therefore their left sides must be superkeys. Thus, `Movies2` is in BCNF.  $\square$

**Example 3.17:** We claim that any two-attribute relation is in BCNF. We need to examine the possible nontrivial FD's with a single attribute on the right. There are not too many cases to consider, so let us consider them in turn. In what follows, suppose that the attributes are  $A$  and  $B$ .

1. There are no nontrivial FD's. Then surely the BCNF condition must hold, because only a nontrivial FD can violate this condition. Incidentally, note that  $\{A, B\}$  is the only key in this case.
2.  $A \rightarrow B$  holds, but  $B \rightarrow A$  does not hold. In this case,  $A$  is the only key, and each nontrivial FD contains  $A$  on the left (in fact the left can only be  $A$ ). Thus there is no violation of the BCNF condition.
3.  $B \rightarrow A$  holds, but  $A \rightarrow B$  does not hold. This case is symmetric to case (2).
4. Both  $A \rightarrow B$  and  $B \rightarrow A$  hold. Then both  $A$  and  $B$  are keys. Surely any FD has at least one of these on the left, so there can be no BCNF violation.

It is worth noticing from case (4) above that there may be more than one key for a relation. Further, the BCNF condition only requires that *some* key be contained in the left side of any nontrivial FD, not that all keys are contained in the left side. Also observe that a relation with two attributes, each functionally determining the other, is not completely implausible. For example, a company may assign its employees unique employee ID's and also record their Social Security numbers. A relation with attributes `empID` and `ssNo` would have each attribute functionally determining the other. Put another way, each attribute is a key, since we don't expect to find two tuples that agree on either attribute.  $\square$

### 3.3.4 Decomposition into BCNF

By repeatedly choosing suitable decompositions, we can break any relation schema into a collection of subsets of its attributes with the following important properties:

1. These subsets are the schemas of relations in BCNF.
2. The data in the original relation is represented faithfully by the data in the relations that are the result of the decomposition, in a sense to be made precise in Section 3.4.1. Roughly, we need to be able to reconstruct the original relation instance exactly from the decomposed relation instances.

Example 3.17 suggests that perhaps all we have to do is break a relation schema into two-attribute subsets, and the result is surely in BCNF. However, such an arbitrary decomposition will not satisfy condition (2), as we shall see in Section 3.4.1. In fact, we must be more careful and use the violating FD's to guide our decomposition.

The decomposition strategy we shall follow is to look for a nontrivial FD  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  that violates BCNF; i.e.,  $\{A_1, A_2, \dots, A_n\}$  is not a superkey. We shall add to the right side as many attributes as are functionally determined by  $\{A_1, A_2, \dots, A_n\}$ . This step is not mandatory, but it often reduces the total amount of work done, and we shall include it in our algorithm. Figure 3.8 illustrates how the attributes are broken into two overlapping relation schemas. One is all the attributes involved in the violating FD, and the other is the left side of the FD plus all the attributes *not* involved in the FD, i.e., all the attributes except those  $B$ 's that are not  $A$ 's.

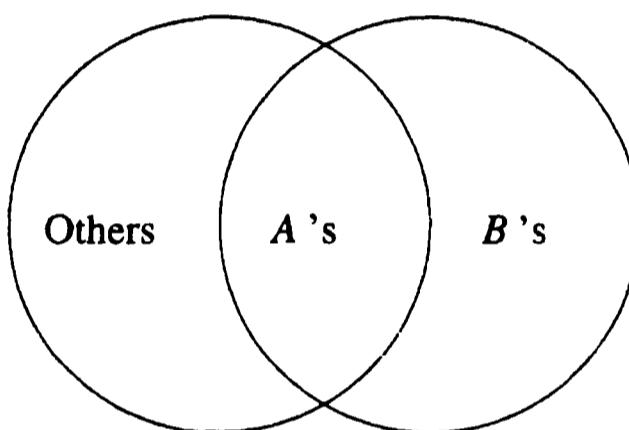


Figure 3.8: Relation schema decomposition based on a BCNF violation

**Example 3.18:** Consider our running example, the `Movies1` relation of Fig. 3.6. We saw in Example 3.15 that

$$\text{title year} \rightarrow \text{length genre studioName}$$

is a BCNF violation. In this case, the right side already includes all the attributes functionally determined by `title` and `year`, so we shall use this BCNF violation to decompose `Movies1` into:

1. The schema  $\{\text{title}, \text{year}, \text{length}, \text{genre}, \text{studioName}\}$  consisting of all the attributes on either side of the FD.
2. The schema  $\{\text{title}, \text{year}, \text{starName}\}$  consisting of the left side of the FD plus all attributes of `Movies1` that do not appear in either side of the FD (only `starName`, in this case).

Notice that these schemas are the ones selected for relations `Movies2` and `Movies3` in Example 3.14. We observed in Example 3.16 that `Movies2` is in BCNF. `Movies3` is also in BCNF; it has no nontrivial FD's.  $\square$

In Example 3.18, one judicious application of the decomposition rule is enough to produce a collection of relations that are in BCNF. In general, that is not the case, as the next example shows.

**Example 3.19:** Consider a relation with schema

$$\{\text{title}, \text{year}, \text{studioName}, \text{president}, \text{presAddr}\}$$

That is, each tuple of this relation tells about a movie, its studio, the president of the studio, and the address of the president of the studio. Three FD's that we would assume in this relation are

$$\begin{aligned} \text{title year} &\rightarrow \text{studioName} \\ \text{studioName} &\rightarrow \text{president} \\ \text{president} &\rightarrow \text{presAddr} \end{aligned}$$

By closing sets of these five attributes, we discover that  $\{\text{title}, \text{year}\}$  is the only key for this relation. Thus the last two FD's above violate BCNF. Suppose we choose to decompose starting with

$$\text{studioName} \rightarrow \text{president}$$

First, we add to the right side of this functional dependency any other attributes in the closure of **studioName**. That closure includes **presAddr**, so our final choice of FD for the decomposition is:

$$\text{studioName} \rightarrow \text{president presAddr}$$

The decomposition based on this FD yields the following two relation schemas.

$$\begin{aligned} \{\text{title}, \text{year}, \text{studioName}\} \\ \{\text{studioName}, \text{president}, \text{presAddr}\} \end{aligned}$$

If we use Algorithm 3.12 to project FD's, we determine that the FD's for the first relation has a basis:

$$\text{title year} \rightarrow \text{studioName}$$

while the second has:

$$\begin{aligned} \text{studioName} &\rightarrow \text{president} \\ \text{president} &\rightarrow \text{presAddr} \end{aligned}$$

The sole key for the first relation is  $\{\text{title}, \text{year}\}$ , and it is therefore in BCNF. However, the second has  $\{\text{studioName}\}$  for its only key but also has the FD:

$$\text{president} \rightarrow \text{presAddr}$$

which is a BCNF violation. Thus, we must decompose again, this time using the above FD. The resulting three relation schemas, all in BCNF, are:

```

{title, year, studioName}
{studioName, president}
{president, presAddr}

```

□

In general, we must keep applying the decomposition rule as many times as needed, until all our relations are in BCNF. We can be sure of ultimate success, because every time we apply the decomposition rule to a relation  $R$ , the two resulting schemas each have fewer attributes than that of  $R$ . As we saw in Example 3.17, when we get down to two attributes, the relation is sure to be in BCNF; often relations with larger sets of attributes are also in BCNF. The strategy is summarized below.

**Algorithm 3.20:** BCNF Decomposition Algorithm.

**INPUT:** A relation  $R_0$  with a set of functional dependencies  $S_0$ .

**OUTPUT:** A decomposition of  $R_0$  into a collection of relations, all of which are in BCNF.

**METHOD:** The following steps can be applied recursively to any relation  $R$  and set of FD's  $S$ . Initially, apply them with  $R = R_0$  and  $S = S_0$ .

1. Check whether  $R$  is in BCNF. If so, nothing more needs to be done. Return  $\{R\}$  as the answer.
2. If there are BCNF violations, let one be  $X \rightarrow Y$ . Use Algorithm 3.7 to compute  $X^+$ . Choose  $R_1 = X^+$  as one relation schema and let  $R_2$  have attributes  $X$  and those attributes of  $R$  that are not in  $X^+$ .
3. Use Algorithm 3.12 to compute the sets of FD's for  $R_1$  and  $R_2$ ; let these be  $S_1$  and  $S_2$ , respectively.
4. Recursively decompose  $R_1$  and  $R_2$  using this algorithm. Return the union of the results of these decompositions.

□

### 3.3.5 Exercises for Section 3.3

**Exercise 3.3.1:** For each of the following relation schemas and sets of FD's:

- a)  $R(A, B, C, D)$  with FD's  $AB \rightarrow C$ ,  $C \rightarrow D$ , and  $D \rightarrow A$ .
- b)  $R(A, B, C, D)$  with FD's  $B \rightarrow C$  and  $B \rightarrow D$ .
- c)  $R(A, B, C, D)$  with FD's  $AB \rightarrow C$ ,  $BC \rightarrow D$ ,  $CD \rightarrow A$ , and  $AD \rightarrow B$ .
- d)  $R(A, B, C, D)$  with FD's  $A \rightarrow B$ ,  $B \rightarrow C$ ,  $C \rightarrow D$ , and  $D \rightarrow A$ .

- e)  $R(A, B, C, D, E)$  with FD's  $AB \rightarrow C$ ,  $DE \rightarrow C$ , and  $B \rightarrow D$ .
- f)  $R(A, B, C, D, E)$  with FD's  $AB \rightarrow C$ ,  $C \rightarrow D$ ,  $D \rightarrow B$ , and  $D \rightarrow E$ .

do the following:

- i) Indicate all the BCNF violations. Do not forget to consider FD's that are not in the given set, but follow from them. However, it is not necessary to give violations that have more than one attribute on the right side.
- ii) Decompose the relations, as necessary, into collections of relations that are in BCNF.

**Exercise 3.3.2:** We mentioned in Section 3.3.4 that we would exercise our option to expand the right side of an FD that is a BCNF violation if possible. Consider a relation  $R$  whose schema is the set of attributes  $\{A, B, C, D\}$  with FD's  $A \rightarrow B$  and  $A \rightarrow C$ . Either is a BCNF violation, because the only key for  $R$  is  $\{A, D\}$ . Suppose we begin by decomposing  $R$  according to  $A \rightarrow B$ . Do we ultimately get the same result as if we first expand the BCNF violation to  $A \rightarrow BC$ ? Why or why not?

! **Exercise 3.3.3:** Let  $R$  be as in Exercise 3.3.2, but let the FD's be  $A \rightarrow B$  and  $B \rightarrow C$ . Again compare decomposing using  $A \rightarrow B$  first against decomposing by  $A \rightarrow BC$  first.

! **Exercise 3.3.4:** Suppose we have a relation schema  $R(A, B, C)$  with FD  $A \rightarrow B$ . Suppose also that we decide to decompose this schema into  $S(A, B)$  and  $T(B, C)$ . Give an example of an instance of relation  $R$  whose projection onto  $S$  and  $T$  and subsequent rejoining as in Section 3.4.1 does not yield the same relation instance. That is,  $\pi_{A,B}(R) \bowtie \pi_{B,C}(R) \neq R$ .

## 3.4 Decomposition: The Good, Bad, and Ugly

So far, we observed that before we decompose a relation schema into BCNF, it can exhibit anomalies; after we decompose, the resulting relations do not exhibit anomalies. That's the "good." But decomposition can also have some bad, if not downright ugly, consequences. In this section, we shall consider three distinct properties we would like a decomposition to have.

1. *Elimination of Anomalies* by decomposition as in Section 3.3.
2. *Recoverability of Information*. Can we recover the original relation from the tuples in its decomposition?
3. *Preservation of Dependencies*. If we check the projected FD's in the relations of the decomposition, can we be sure that when we reconstruct the original relation from the decomposition by joining, the result will satisfy the original FD's?

It turns out that the BCNF decomposition of Algorithm 3.20 gives us (1) and (2), but does not necessarily give us all three. In Section 3.5 we shall see another way to pick a decomposition that gives us (2) and (3) but does not necessarily give us (1). In fact, there is no way to get all three at once.

### 3.4.1 Recovering Information from a Decomposition

Since we learned that every two-attribute relation is in BCNF, why did we have to go through the trouble of Algorithm 3.20? Why not just take any relation  $R$  and decompose it into relations, each of whose schemas is a pair of  $R$ 's attributes? The answer is that the data in the decomposed relations, even if their tuples were each the projection of a relation instance of  $R$ , might not allow us to join the relations of the decomposition and get the instance of  $R$  back. If we do get  $R$  back, then we say the decomposition has a *lossless join*.

However, if we decompose using Algorithm 3.20, where all decompositions are motivated by a BCNF-violating FD, then the projections of the original tuples can be joined again to produce all and only the original tuples. We shall consider why here. Then, in Section 3.4.2 we shall give an algorithm called the “chase,” for testing whether the projection of a relation onto any decomposition allows us to recover the relation by rejoining.

To simplify the situation, consider a relation  $R(A, B, C)$  and an FD  $B \rightarrow C$  that is a BCNF violation. The decomposition based on the FD  $B \rightarrow C$  separates the attributes into relations  $R_1(A, B)$  and  $R_2(B, C)$ .

Let  $t$  be a tuple of  $R$ . We may write  $t = (a, b, c)$ , where  $a$ ,  $b$ , and  $c$  are the components of  $t$  for attributes  $A$ ,  $B$ , and  $C$ , respectively. Tuple  $t$  projects as  $(a, b)$  in  $R_1(A, B) = \pi_{A,B}(R)$  and as  $(b, c)$  in  $R_2(B, C) = \pi_{B,C}(R)$ . When we compute the natural join  $R_1 \bowtie R_2$ , these two projected tuples join, because they agree on the common  $B$  component (they both have  $b$  there). They give us  $t = (a, b, c)$ , the tuple we started with, in the join. That is, regardless of what tuple  $t$  we started with, we can always join its projections to get  $t$  back.

However, getting back those tuples we started with is not enough to assure that the original relation  $R$  is truly represented by the decomposition. Consider what happens if there are two tuples of  $R$ , say  $t = (a, b, c)$  and  $v = (d, b, e)$ . When we project  $t$  onto  $R_1(A, B)$  we get  $u = (a, b)$ , and when we project  $v$  onto  $R_2(B, C)$  we get  $w = (b, e)$ . These tuples also match in the natural join, and the resulting tuple is  $x = (a, b, e)$ . Is it possible that  $x$  is a bogus tuple? That is, could  $(a, b, e)$  not be a tuple of  $R$ ?

Since we assume the FD  $B \rightarrow C$  for relation  $R$ , the answer is “no.” Recall that this FD says any two tuples of  $R$  that agree in their  $B$  components must also agree in their  $C$  components. Since  $t$  and  $v$  agree in their  $B$  components, they also agree on their  $C$  components. That means  $c = e$ ; i.e., the two values we supposed were different are really the same. Thus, tuple  $(a, b, e)$  of  $R$  is really  $(a, b, c)$ ; that is,  $x = t$ .

Since  $t$  is in  $R$ , it must be that  $x$  is in  $R$ . Put another way, as long as FD  $B \rightarrow C$  holds, the joining of two projected tuples cannot produce a bogus tuple

Rather, every tuple produced by the natural join is guaranteed to be a tuple of  $R$ .

This argument works in general. We assumed  $A$ ,  $B$ , and  $C$  were each single attributes, but the same argument would apply if they were any sets of attributes  $X$ ,  $Y$  and  $Z$ . That is, if  $Y \rightarrow Z$  holds in  $R$ , whose attributes are  $X \cup Y \cup Z$ , then  $R = \pi_{X \cup Y}(R) \bowtie \pi_{Y \cup Z}(R)$ .

We may conclude:

- If we decompose a relation according to Algorithm 3.20, then the original relation can be recovered exactly by the natural join.

To see why, we argued above that at any one step of the recursive decomposition, a relation is equal to the join of its projections onto the two components. If those components are decomposed further, they can also be recovered by the natural join from their decomposed relations. Thus, an easy induction on the number of binary decomposition steps says that the original relation is always the natural join of whatever relations it is decomposed into. We can also prove that the natural join is associative and commutative, so the order in which we perform the natural join of the decomposition components does not matter.

The FD  $Y \rightarrow Z$ , or its symmetric FD  $Y \rightarrow X$ , is essential. Without one of these FD's, we might not be able to recover the original relation. Here is an example.

**Example 3.21:** Suppose we have the relation  $R(A, B, C)$  as above, but neither of the FD's  $B \rightarrow A$  nor  $B \rightarrow C$  holds. Then  $R$  might consist of the two tuples

$A$	$B$	$C$
1	2	3
4	2	5

The projections of  $R$  onto the relations with schemas  $\{A, B\}$  and  $\{B, C\}$  are  $R_1 = \pi_{AB}(R) =$

$A$	$B$
1	2
4	2

and  $R_2 = \pi_{BC}(R) =$

$B$	$C$
2	3
2	5

respectively. Since all four tuples share the same  $B$ -value, 2, each tuple of one relation joins with both tuples of the other relation. When we try to reconstruct  $R$  by the natural join of the projected relations, we get  $R_3 = R_1 \bowtie R_2 =$

## Is Join the Only Way to Recover?

We have assumed that the only possible way we could reconstruct a relation from its projections is to use the natural join. However, might there be some other algorithm to reconstruct the original relation that would work even in cases where the natural join fails? There is in fact no such other way. In Example 3.21, the relations  $R$  and  $R_3$  are different instances, yet have exactly the same projections onto  $\{A, B\}$  and  $\{B, C\}$ , namely the instances we called  $R_1$  and  $R_2$ , respectively. Thus, given  $R_1$  and  $R_2$ , no algorithm whatsoever can tell whether the original instance was  $R$  or  $R_3$ .

Moreover, this example is not unusual. Given any decomposition of a relation with attributes  $X \cup Y \cup Z$  into relations with schemas  $X \cup Y$  and  $Y \cup Z$ , where neither  $Y \rightarrow X$  nor  $Y \rightarrow Z$  holds, we can construct an example similar to Example 3.21 where the original instance cannot be determined from its projections.

$A$	$B$	$C$
1	2	3
1	2	5
4	2	3
4	2	5

That is, we get “too much”; we get two bogus tuples,  $(1, 2, 5)$  and  $(4, 2, 3)$ , that were not in the original relation  $R$ .  $\square$

### 3.4.2 The Chase Test for Lossless Join

In Section 3.4.1 we argued why a particular decomposition, that of  $R(A, B, C)$  into  $\{A, B\}$  and  $\{B, C\}$ , with a particular FD,  $B \rightarrow C$ , had a lossless join. Now, consider a more general situation. We have decomposed relation  $R$  into relations with sets of attributes  $S_1, S_2, \dots, S_k$ . We have a given set of FD’s  $F$  that hold in  $R$ . Is it true that if we project  $R$  onto the relations of the decomposition, then we can recover  $R$  by taking the natural join of all these relations? That is, is it true that  $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \dots \bowtie \pi_{S_k}(R) = R$ ? Three important things to remember are:

- The natural join is associative and commutative. It does not matter in what order we join the projections; we shall get the same relation as a result. In particular, the result is the set of tuples  $t$  such that for all  $i = 1, 2, \dots, k$ ,  $t$  projected onto the set of attributes  $S_i$  is a tuple in  $\pi_{S_i}(R)$ .

- Any tuple  $t$  in  $R$  is surely in  $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$ . The reason is that the projection of  $t$  onto  $S_i$  is surely in  $\pi_{S_i}(R)$  for each  $i$ , and therefore by our first point above,  $t$  is in the result of the join.
- As a consequence,  $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R) = R$  when the FD's in  $F$  hold for  $R$  if and only if every tuple in the join is also in  $R$ . That is, the membership test is all we need to verify that the decomposition has a lossless join.

The *chase* test for a lossless join is just an organized way to see whether a tuple  $t$  in  $\pi_{S_1}(R) \bowtie \pi_{S_2}(R) \bowtie \cdots \bowtie \pi_{S_k}(R)$  can be proved, using the FD's in  $F$ , also to be a tuple in  $R$ . If  $t$  is in the join, then there must be tuples in  $R$ , say  $t_1, t_2, \dots, t_k$ , such that  $t$  is the join of the projections of each  $t_i$  onto the set of attributes  $S_i$ , for  $i = 1, 2, \dots, k$ . We therefore know that  $t_i$  agrees with  $t$  on the attributes of  $S_i$ , but  $t_i$  has unknown values in its components not in  $S_i$ .

We draw a picture of what we know, called a *tableau*. Assuming  $R$  has attributes  $A, B, \dots$  we use  $a, b, \dots$  for the components of  $t$ . For  $t_i$ , we use the same letter as  $t$  in the components that are in  $S_i$ , but we subscript the letter with  $i$  if the component is not in  $i$ . In that way,  $t_i$  will agree with  $t$  for the attributes of  $S_i$ , but have a unique value — one that can appear nowhere else in the tableau — for other attributes.

**Example 3.22:** Suppose we have relation  $R(A, B, C, D)$ , which we have decomposed into relations with sets of attributes  $S_1 = \{A, D\}$ ,  $S_2 = \{A, C\}$ , and  $S_3 = \{B, C, D\}$ . Then the tableau for this decomposition is shown in Fig. 3.9.

$A$	$B$	$C$	$D$
$a$	$b_1$	$c_1$	$d$
$a$	$b_2$	$c$	$d_2$
$a_3$	$b$	$c$	$d$

Figure 3.9: Tableau for the decomposition of  $R$  into  $\{A, D\}$ ,  $\{A, C\}$ , and  $\{B, C, D\}$

The first row corresponds to set of attributes  $A$  and  $D$ . Notice that the components for attributes  $A$  and  $D$  are the unsubscripted letters  $a$  and  $d$ . However, for the other attributes,  $b$  and  $c$ , we add the subscript 1 to indicate that they are arbitrary values. This choice makes sense, since the tuple  $(a, b_1, c_1, d)$  represents a tuple of  $R$  that contributes to  $t = (a, b, c, d)$  by being projected onto  $\{A, D\}$  and then joined with other tuples. Since the  $B$ - and  $C$ -components of this tuple are projected out, we know nothing yet about what values the tuple had for those attributes.

Similarly, the second row has the unsubscripted letters in attributes  $A$  and  $C$ , while the subscript 2 is used for the other attributes. The last row has the unsubscripted letters in components for  $\{B, C, D\}$  and subscript 3 on  $a$ . Since

each row uses its own number as a subscript, the only symbols that can appear more than once are the unsubscripted letters.  $\square$

Remember that our goal is to use the given set of FD's  $F$  to prove that  $t$  is really in  $R$ . In order to do so, we "chase" the tableau by applying the FD's in  $F$  to equate symbols in the tableau whenever we can. If we discover that one of the rows is actually the same as  $t$  (that is, the row becomes all unsubscripted symbols), then we have proved that any tuple  $t$  in the join of the projections was actually a tuple of  $R$ .

To avoid confusion, when equating two symbols, if one of them is unsubscripted, make the other be the same. However, if we equate two symbols, both with their own subscript, then you can change either to be the other. However, remember that when equating symbols, you must change all occurrences of one to be the other, not just some of the occurrences.

**Example 3.23:** Let us continue with the decomposition of Example 3.22, and suppose the given FD's are  $A \rightarrow B$ ,  $B \rightarrow C$ , and  $CD \rightarrow A$ . Start with the tableau of Fig. 3.9. Since the first two rows agree in their  $A$ -components, the FD  $A \rightarrow B$  tells us they must also agree in their  $B$ -components. That is,  $b_1 = b_2$ . We can replace either one with the other, since they are both subscripted. Let us replace  $b_2$  by  $b_1$ . Then the resulting tableau is:

$A$	$B$	$C$	$D$
$a$	$b_1$	$c_1$	$d$
$a$	$b_1$	$c$	$d_2$
$a_3$	$b$	$c$	$d$

Now, we see that the first two rows have equal  $B$ -values, and so we may use the FD  $B \rightarrow C$  to deduce that their  $C$ -components,  $c_1$  and  $c$ , are the same. Since  $c$  is unsubscripted, we replace  $c_1$  by  $c$ , leaving:

$A$	$B$	$C$	$D$
$a$	$b_1$	$c$	$d$
$a$	$b_1$	$c$	$d_2$
$a_3$	$b$	$c$	$d$

Next, we observe that the first and third rows agree in both columns  $C$  and  $D$ . Thus, we may apply the FD  $CD \rightarrow A$  to deduce that these rows also have the same  $A$ -value; that is,  $a = a_3$ . We replace  $a_3$  by  $a$ , giving us:

$A$	$B$	$C$	$D$
$a$	$b_1$	$c$	$d$
$a$	$b_1$	$c$	$d_2$
$a$	$b$	$c$	$d$

At this point, we see that the last row has become equal to  $t$ , that is,  $(a, b, c, d)$ . We have proved that if  $R$  satisfies the FD's  $A \rightarrow B$ ,  $B \rightarrow C$ , and  $CD \rightarrow A$ , then whenever we project onto  $\{A, D\}$ ,  $\{A, C\}$ , and  $\{B, C, D\}$  and rejoin, what we get must have been in  $R$ . In particular, what we get is the same as the tuple of  $R$  that we projected onto  $\{B, C, D\}$ .  $\square$

### 3.4.3 Why the Chase Works

There are two issues to address:

1. When the chase results in a row that matches the tuple  $t$  (i.e., the tableau is shown to have a row with all unsubscripted variables), why must the join be lossless?
2. When, after applying FD's whenever we can, we still find no row of all unsubscripted variables, why must the join not be lossless?

Question (1) is easy to answer. The chase process itself is a proof that one of the projected tuples from  $R$  must in fact be the tuple  $t$  that is produced by the join. We also know that every tuple in  $R$  is sure to come back if we project and join. Thus, the chase has proved that the result of projection and join is exactly  $R$ .

For the second question, suppose that we eventually derive a tableau without an unsubscripted row, and that this tableau does not allow us to apply any of the FD's to equate any symbols. Then think of the tableau as an instance of the relation  $R$ . It obviously satisfies the given FD's, because none can be applied to equate symbols. We know that the  $i$ th row has unsubscripted symbols in the attributes of  $S_i$ , the  $i$ th relation of the decomposition. Thus, when we project this relation onto the  $S_i$ 's and take the natural join, we get the tuple with all unsubscripted variables. This tuple is not in  $R$ , so we conclude that the join is not lossless.

**Example 3.24:** Consider the relation  $R(A, B, C, D)$  with the FD  $B \rightarrow AD$  and the proposed decomposition  $\{A, B\}$ ,  $\{B, C\}$ , and  $\{C, D\}$ . Here is the initial tableau:

$A$	$B$	$C$	$D$
$a$	$b$	$c_1$	$d_1$
$a_2$	$b$	$c$	$d_2$
$a_3$	$b_3$	$c$	$d$

When we apply the lone FD, we deduce that  $a = a_2$  and  $d_1 = d_2$ . Thus, the final tableau is:

$A$	$B$	$C$	$D$
$a$	$b$	$c_1$	$d_1$
$a$	$b$	$c$	$d_1$
$a_3$	$b_3$	$c$	$d$

No more changes can be made because of the given FD's, and there is no row that is fully unsubscripted. Thus, this decomposition does not have a lossless join. We can verify that fact by treating the above tableau as a relation with three tuples. When we project onto  $\{A, B\}$ , we get  $\{(a, b)\}, (a_3, b_3)\}$ . The projection onto  $\{B, C\}$  is  $\{(b, c_1), (b, c), (b_3, c)\}$ , and the projection onto  $\{C, D\}$  is  $(c_1, d_1), (c, d_1), (c, d)\}$ . If we join the first two projections, we get  $\{(a, b, c_1), (a, b, c), (a_3, b_3, c)\}$ . Joining this relation with the third projection gives  $\{(a, b, c_1, d_1), (a, b, c, d_1), (a, b, c, d), (a_3, b_3, c, d_1), (a_3, b_3, c, d)\}$ . Notice that this join has two more tuples than  $R$ , and in particular it has the tuple  $(a, b, c, d)$ , as it must.  $\square$

### 3.4.4 Dependency Preservation

We mentioned that it is not possible, in some cases, to decompose a relation into BCNF relations that have both the lossless-join and dependency-preservation properties. Below is an example where we need to make a tradeoff between preserving dependencies and BCNF.

**Example 3.25:** Suppose we have a relation **Bookings** with attributes:

1. **title**, the name of a movie.
2. **theater**, the name of a theater where the movie is being shown.
3. **city**, the city where the theater is located.

The intent behind a tuple  $(m, t, c)$  is that the movie with title  $m$  is currently being shown at theater  $t$  in city  $c$ .

We might reasonably assert the following FD's:

$$\begin{aligned} \text{theater} &\rightarrow \text{city} \\ \text{title } \text{city} &\rightarrow \text{theater} \end{aligned}$$

The first says that a theater is located in one city. The second is not obvious but is based on the common practice of not booking a movie into two theaters in the same city. We shall assert this FD if only for the sake of the example.

Let us first find the keys. No single attribute is a key. For example, **title** is not a key because a movie can play in several theaters at once and in several cities at once.<sup>2</sup> Also, **theater** is not a key, because although **theater** functionally determines **city**, there are multiscreen theaters that show many movies at once. Thus, **theater** does not determine **title**. Finally, **city** is not a key because cities usually have more than one theater and more than one movie playing.

---

<sup>2</sup>In this example we assume that there are not two “current” movies with the same title, even though we have previously recognized that there could be two movies with the same title made in different years.

On the other hand, two of the three sets of two attributes are keys. Clearly  $\{\text{title}, \text{city}\}$  is a key because of the given FD that says these attributes functionally determine **theater**.

It is also true that  $\{\text{theater}, \text{title}\}$  is a key, because its closure includes **city** due to the given FD  $\text{theater} \rightarrow \text{city}$ . The remaining pair of attributes, **city** and **theater**, do not functionally determine **title**, because of multiscreen theaters, and are therefore not a key. We conclude that the only two keys are

$$\begin{aligned} &\{\text{title}, \text{city}\} \\ &\{\text{theater}, \text{title}\} \end{aligned}$$

Now we immediately see a BCNF violation. We were given functional dependency  $\text{theater} \rightarrow \text{city}$ , but its left side, **theater**, is not a superkey. We are therefore tempted to decompose, using this BCNF-violating FD, into the two relation schemas:

$$\begin{aligned} &\{\text{theater}, \text{city}\} \\ &\{\text{theater}, \text{title}\} \end{aligned}$$

There is a problem with this decomposition, concerning the FD

$$\text{title city} \rightarrow \text{theater}$$

There could be current relations for the decomposed schemas that satisfy the FD  $\text{theater} \rightarrow \text{city}$  (which can be checked in the relation  $\{\text{theater}, \text{city}\}$ ) but that, when joined, yield a relation not satisfying  $\text{title city} \rightarrow \text{theater}$ . For instance, the two relations

<i>theater</i>	<i>city</i>
Guild	Menlo Park
Park	Menlo Park

and

<i>theater</i>	<i>title</i>
Guild	Antz
Park	Antz

are permissible according to the FD's that apply to each of the above relations, but when we join them we get two tuples

<i>theater</i>	<i>city</i>	<i>title</i>
Guild	Menlo Park	Antz
Park	Menlo Park	Antz

that violate the FD  $\text{title city} \rightarrow \text{theater}$ .  $\square$

### 3.4.5 Exercises for Section 3.4

**Exercise 3.4.1:** Let  $R(A, B, C, D, E)$  be decomposed into relations with the following three sets of attributes:  $\{A, B, C\}$ ,  $\{B, C, D\}$ , and  $\{A, C, E\}$ . For each of the following sets of FD's, use the chase test to tell whether the decomposition of  $R$  is lossless. For those that are not lossless, give an example of an instance of  $R$  that returns more than  $R$  when projected onto the decomposed relations and rejoined.

- a)  $B \rightarrow E$  and  $CE \rightarrow A$ .
- b)  $AC \rightarrow E$  and  $BC \rightarrow D$ .
- c)  $A \rightarrow D$ ,  $D \rightarrow E$ , and  $B \rightarrow D$ .
- d)  $A \rightarrow D$ ,  $CD \rightarrow E$ , and  $E \rightarrow D$ .

! **Exercise 3.4.2:** For each of the sets of FD's in Exercise 3.4.1, are dependencies preserved by the decomposition?

## 3.5 Third Normal Form

The solution to the problem illustrated by Example 3.25 is to relax our BCNF requirement slightly, in order to allow the occasional relation schema that cannot be decomposed into BCNF relations without losing the ability to check the FD's. This relaxed condition is called “third normal form.” In this section we shall give the requirements for third normal form, and then show how to do a decomposition in a manner quite different from Algorithm 3.20, in order to obtain relations in third normal form that have both the lossless-join and dependency-preservation properties.

### 3.5.1 Definition of Third Normal Form

A relation  $R$  is in *third normal form* (3NF) if:

- Whenever  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  is a nontrivial FD, either

$$\{A_1, A_2, \dots, A_n\}$$

is a superkey, or those of  $B_1, B_2, \dots, B_m$  that are not among the  $A$ 's, are each a member of some key (not necessarily the same key).

An attribute that is a member of some key is often said to be *prime*. Thus, the 3NF condition can be stated as “for each nontrivial FD, either the left side is a superkey, or the right side consists of prime attributes only.”

Note that the difference between this 3NF condition and the BCNF condition is the clause “is a member of some key (i.e., prime).” This clause “excuses” an FD like `theater → city` in Example 3.25, because the right side, `city`, is prime.

## Other Normal Forms

If there is a “third normal form,” what happened to the first two “normal forms”? They indeed were defined, but today there is little use for them. *First normal form* is simply the condition that every component of every tuple is an atomic value. *Second normal form* is a less restrictive version of 3NF. There is also a “fourth normal form” that we shall meet in Section 3.6.

### 3.5.2 The Synthesis Algorithm for 3NF Schemas

We can now explain and justify how we decompose a relation  $R$  into a set of relations such that:

- a) The relations of the decomposition are all in 3NF.
- b) The decomposition has a lossless join.
- c) The decomposition has the dependency-preservation property.

**Algorithm 3.26:** Synthesis of Third-Normal-Form Relations With a Lossless Join and Dependency Preservation.

**INPUT:** A relation  $R$  and a set  $F$  of functional dependencies that hold for  $R$ .

**OUTPUT:** A decomposition of  $R$  into a collection of relations, each of which is in 3NF. The decomposition has the lossless-join and dependency-preservation properties.

**METHOD:** Perform the following steps:

1. Find a minimal basis for  $F$ , say  $G$ .
2. For each functional dependency  $X \rightarrow A$  in  $G$ , use  $XA$  as the schema of one of the relations in the decomposition.
3. If none of the sets of relations from Step 2 is a superkey for  $R$ , add another relation whose schema is a key for  $R$ .

□

**Example 3.27:** Consider the relation  $R(A, B, C, D, E)$  with FD’s  $AB \rightarrow C$ ,  $C \rightarrow B$ , and  $A \rightarrow D$ . To start, notice that the given FD’s are their own minimal basis. To check, we need to do a bit of work. First, we need to verify that we cannot eliminate any of the given dependencies. That is, we show, using Algorithm 3.7, that no two of the FD’s imply the third. For example, we must take the closure of  $\{A, B\}$ , the left side of the first FD, using only the

second and third FD's,  $C \rightarrow B$  and  $A \rightarrow D$ . This closure includes  $D$  but not  $C$ , so we conclude that the first FD  $AB \rightarrow C$  is not implied by the second and third FD's. We get a similar conclusion if we try to drop the second or third FD.

We must also verify that we cannot eliminate any attributes from a left side. In this simple case, the only possibility is that we could eliminate  $A$  or  $B$  from the first FD. For example, if we eliminate  $A$ , we would be left with  $B \rightarrow C$ . We must show that  $B \rightarrow C$  is not implied by the three original FD's,  $AB \rightarrow C$ ,  $C \rightarrow B$ , and  $A \rightarrow D$ . With these FD's, the closure of  $\{B\}$  is just  $B$ , so  $B \rightarrow C$  does not follow. A similar conclusion is drawn if we try to drop  $B$  from  $AB \rightarrow C$ . Thus, we have our minimal basis.

We start the 3NF synthesis by taking the attributes of each FD as a relation schema. That is, we get relations  $S_1(A, B, C)$ ,  $S_2(B, C)$ , and  $S_3(A, D)$ . It is never necessary to use a relation whose schema is a proper subset of another relation's schema, so we can drop  $S_2$ .

We must also consider whether we need to add a relation whose schema is a key. In this example,  $R$  has two keys:  $\{A, B, E\}$  and  $\{A, C, E\}$ , as you can verify. Neither of these keys is a subset of the schemas chosen so far. Thus, we must add one of them, say  $S_4(A, B, E)$ . The final decomposition of  $R$  is thus  $S_1(A, B, C)$ ,  $S_3(A, D)$ , and  $S_4(A, B, E)$ .  $\square$

### 3.5.3 Why the 3NF Synthesis Algorithm Works

We need to show three things: that the lossless-join and dependency-preservation properties hold, and that all the relations of the decomposition are in 3NF.

1. *Lossless Join.* Start with a relation of the decomposition whose set of attributes  $K$  is a superkey. Consider the sequence of FD's that are used in Algorithm 3.7 to expand  $K$  to become  $K^+$ . Since  $K$  is a superkey, we know  $K^+$  is all the attributes. The same sequence of FD applications on the tableau cause the subscripted symbols in the row corresponding to  $K$  to be equated to unsubscripted symbols in the same order as the attributes were added to the closure. Thus, the chase test concludes that the decomposition is lossless.
2. *Dependency Preservation.* Each FD of the minimal basis has all its attributes in some relation of the decomposition. Thus, each dependency can be checked in the decomposed relations.
3. *Third Normal Form.* If we have to add a relation whose schema is a key, then this relation is surely in 3NF. The reason is that all attributes of this relation are prime, and thus no violation of 3NF could be present in this relation. For the relations whose schemas are derived from the FD's of a minimal basis, the proof that they are in 3NF is beyond the scope of this book. The argument involves showing that a 3NF violation implies that the basis is not minimal.

### 3.5.4 Exercises for Section 3.5

**Exercise 3.5.1:** For each of the relation schemas and sets of FD's of Exercise 3.3.1:

- i) Indicate all the 3NF violations.
- ii) Decompose the relations, as necessary, into collections of relations that are in 3NF.

**Exercise 3.5.2:** Consider the relation  $\text{Courses}(C, T, H, R, S, G)$ , whose attributes may be thought of informally as course, teacher, hour, room, student, and grade. Let the set of FD's for  $\text{Courses}$  be  $C \rightarrow T$ ,  $HR \rightarrow C$ ,  $HT \rightarrow R$ ,  $HS \rightarrow R$ , and  $CS \rightarrow G$ . Intuitively, the first says that a course has a unique teacher, and the second says that only one course can meet in a given room at a given hour. The third says that a teacher can be in only one room at a given hour, and the fourth says the same about students. The last says that students get only one grade in a course.

- a) What are all the keys for  $\text{Courses}$ ?
- b) Verify that the given FD's are their own minimal basis.
- c) Use the 3NF synthesis algorithm to find a lossless-join, dependency-preserving decomposition of  $R$  into 3NF relations. Are any of the relations not in BCNF?

**Exercise 3.5.3:** Consider a relation  $\text{Stocks}(B, O, I, S, Q, D)$ , whose attributes may be thought of informally as broker, office (of the broker), investor, stock, quantity (of the stock owned by the investor), and dividend (of the stock). Let the set of FD's for  $\text{Stocks}$  be  $S \rightarrow D$ ,  $I \rightarrow B$ ,  $IS \rightarrow Q$ , and  $B \rightarrow O$ . Repeat Exercise 3.5.2 for the relation  $\text{Stocks}$ .

**Exercise 3.5.4:** Verify, using the chase, that the decomposition of Example 3.27 has a lossless join.

!! **Exercise 3.5.5:** Suppose we modified Algorithm 3.20 (BCNF decomposition) so that instead of decomposing a relation  $R$  whenever  $R$  was not in BCNF, we only decomposed  $R$  if it was not in 3NF. Provide a counterexample to show that this modified algorithm would not necessarily produce a 3NF decomposition with dependency preservation.

## 3.6 Multivalued Dependencies

A “multivalued dependency” is an assertion that two attributes or sets of attributes are independent of one another. This condition is, as we shall see, a generalization of the notion of a functional dependency, in the sense that

every FD implies the corresponding multivalued dependency. However, there are some situations involving independence of attribute sets that cannot be explained as FD's. In this section we shall explore the cause of multivalued dependencies and see how they can be used in database schema design.

### 3.6.1 Attribute Independence and Its Consequent Redundancy

There are occasional situations where we design a relation schema and find it is in BCNF, yet the relation has a kind of redundancy that is not related to FD's. The most common source of redundancy in BCNF schemas is an attempt to put two or more set-valued properties of the key into a single relation.

**Example 3.28:** In this example, we shall suppose that stars may have several addresses, which we break into street and city components. The set of addresses is one of the set-valued properties this relation will store. The second set-valued property of stars that we shall put into this relation is the set of titles and years of movies in which the star appeared. Then Fig. 3.10 is a typical instance of this relation.

<i>name</i>	<i>street</i>	<i>city</i>	<i>title</i>	<i>year</i>
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Malibu	Star Wars	1977
C. Fisher	123 Maple St.	Hollywood	Empire Strikes Back	1980
C. Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980
C. Fisher	123 Maple St.	Hollywood	Return of the Jedi	1983
C. Fisher	5 Locust Ln.	Malibu	Return of the Jedi	1983

Figure 3.10: Sets of addresses independent from movies

We focus in Fig. 3.10 on Carrie Fisher's two hypothetical addresses and her three best-known movies. There is no reason to associate an address with one movie and not another. Thus, the only way to express the fact that addresses and movies are independent properties of stars is to have each address appear with each movie. But when we repeat address and movie facts in all combinations, there is obvious redundancy. For instance, Fig. 3.10 repeats each of Carrie Fisher's addresses three times (once for each of her movies) and each movie twice (once for each address).

Yet there is no BCNF violation in the relation suggested by Fig. 3.10. There are, in fact, no nontrivial FD's at all. For example, attribute *city* is not functionally determined by the other four attributes. There might be a star with two homes that had the same street address in different cities. Then there would be two tuples that agreed in all attributes but *city* and disagreed in *city*. Thus,

**name street title year → city**

is not an FD for our relation. We leave it to the reader to check that none of the five attributes is functionally determined by the other four. Since there are no nontrivial FD's, it follows that all five attributes form the only key and that there are no BCNF violations.  $\square$

### 3.6.2 Definition of Multivalued Dependencies

A *multivalued dependency* (abbreviated MVD) is a statement about some relation  $R$  that when you fix the values for one set of attributes, then the values in certain other attributes are independent of the values of all the other attributes in the relation. More precisely, we say the MVD

$$A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$$

holds for a relation  $R$  if when we restrict ourselves to the tuples of  $R$  that have particular values for each of the attributes among the  $A$ 's, then the set of values we find among the  $B$ 's is independent of the set of values we find among the attributes of  $R$  that are not among the  $A$ 's or  $B$ 's. Still more precisely, we say this MVD holds if

For each pair of tuples  $t$  and  $u$  of relation  $R$  that agree on all the  $A$ 's, we can find in  $R$  some tuple  $v$  that agrees:

1. With both  $t$  and  $u$  on the  $A$ 's,
2. With  $t$  on the  $B$ 's, and
3. With  $u$  on all attributes of  $R$  that are not among the  $A$ 's or  $B$ 's.

Note that we can use this rule with  $t$  and  $u$  interchanged, to infer the existence of a fourth tuple  $w$  that agrees with  $u$  on the  $B$ 's and with  $t$  on the other attributes. As a consequence, for any fixed values of the  $A$ 's, the associated values of the  $B$ 's and the other attributes appear in all possible combinations in different tuples. Figure 3.11 suggests how  $v$  relates to  $t$  and  $u$  when an MVD holds. However, the  $A$ 's and  $B$ 's do not have to appear consecutively.

In general, we may assume that the  $A$ 's and  $B$ 's (left side and right side) of an MVD are disjoint. However, as with FD's, it is permissible to add some of the  $A$ 's to the right side if we wish.

**Example 3.29 :** In Example 3.28 we encountered an MVD that in our notation is expressed:

**name → street city**

That is, for each star's name, the set of addresses appears in conjunction with each of the star's movies. For an example of how the formal definition of this MVD applies, consider the first and fourth tuples from Fig. 3.10:

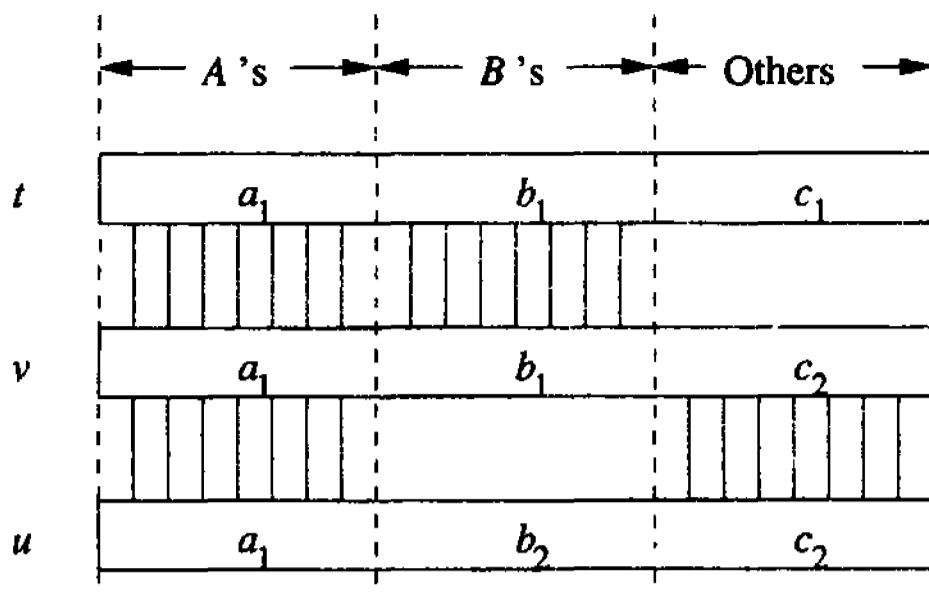


Figure 3.11: A multivalued dependency guarantees that  $v$  exists

<i>name</i>	<i>street</i>	<i>city</i>	<i>title</i>	<i>year</i>
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980

If we let the first tuple be  $t$  and the second be  $u$ , then the MVD asserts that we must also find in  $R$  the tuple that has name C. Fisher, a street and city that agree with the first tuple, and other attributes (*title* and *year*) that agree with the second tuple. There is indeed such a tuple; it is the third tuple of Fig. 3.10.

Similarly, we could let  $t$  be the second tuple above and  $u$  be the first. Then the MVD tells us that there is a tuple of  $R$  that agrees with the second in attributes *name*, *street*, and *city* and with the first in *name*, *title*, and *year*. This tuple also exists; it is the second tuple of Fig. 3.10.  $\square$

### 3.6.3 Reasoning About Multivalued Dependencies

There are a number of rules about MVD's that are similar to the rules we learned for FD's in Section 3.2. For example, MVD's obey

- *Trivial MVD's.* The MVD

$$A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$$

holds in any relation if  $\{B_1, B_2, \dots, B_m\} \subseteq \{A_1, A_2, \dots, A_n\}$ .

- The *transitive rule*, which says that if  $A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$  and  $B_1 B_2 \cdots B_m \rightarrow\!\!\! \rightarrow C_1 C_2 \cdots C_k$  hold for some relation, then so does

$$A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow C_1 C_2 \cdots C_k$$

Any  $C$ 's that are also  $A$ 's may be deleted from the right side.

On the other hand, MVD's do not obey the splitting part of the splitting/combinining rule, as the following example shows.

**Example 3.30:** Consider again Fig. 3.10, where we observed the MVD:

$$\text{name} \rightarrow\!\!\! \rightarrow \text{street city}$$

If the splitting rule applied to MVD's, we would expect

$$\text{name} \rightarrow\!\!\! \rightarrow \text{street}$$

also to be true. This MVD says that each star's street addresses are independent of the other attributes, including *city*. However, that statement is false. Consider, for instance, the first two tuples of Fig. 3.10. The hypothetical MVD would allow us to infer that the tuples with the streets interchanged:

<i>name</i>	<i>street</i>	<i>city</i>	<i>title</i>	<i>year</i>
C. Fisher	5 Locust Ln.	Hollywood	Star Wars	1977
C. Fisher	123 Maple St.	Malibu	Star Wars	1977

were in the relation. But these are not true tuples, because, for instance, the home on 5 Locust Ln. is in Malibu, not Hollywood.  $\square$

However, there are several new rules dealing with MVD's that we can learn.

- *FD Promotion.* Every FD is an MVD. That is, if

$$A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$$

then  $A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$ .

To see why, suppose *R* is some relation for which the FD

$$A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$$

holds, and suppose *t* and *u* are tuples of *R* that agree on the *A*'s. To show that the MVD  $A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$  holds, we have to show that *R* also contains a tuple *v* that agrees with *t* and *u* on the *A*'s, with *t* on the *B*'s, and with *u* on all other attributes. But *v* can be *u*. Surely *u* agrees with *t* and *u* on the *A*'s, because we started by assuming that these two tuples agree on the *A*'s. The FD  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  assures us that *u* agrees with *t* on the *B*'s. And of course *u* agrees with itself on the other attributes. Thus, whenever an FD holds, the corresponding MVD holds.

- *Complementation Rule.* If  $A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$  is an MVD for relation *R*, then *R* also satisfies  $A_1 A_2 \cdots A_n \rightarrow C_1 C_2 \cdots C_k$ , where the *C*'s are all attributes of *R* not among the *A*'s and *B*'s.

That is, swapping the  $B$ 's between two tuples that agree in the  $A$ 's has the same effect as swapping the  $C$ 's.

**Example 3.31:** Again consider the relation of Fig. 3.10, for which we asserted the MVD:

$$\text{name} \rightarrow\!\!\! \rightarrow \text{street city}$$

The complementation rule says that

$$\text{name} \rightarrow\!\!\! \rightarrow \text{title year}$$

must also hold in this relation, because **title** and **year** are the attributes not mentioned in the first MVD. The second MVD intuitively means that each star has a set of movies starred in, which are independent of the star's addresses.  $\square$

An MVD whose right side is a subset of the left side is trivial — it holds in every relation. However, an interesting consequence of the complementation rule is that there are some other MVD's that are trivial, but that look distinctly nontrivial.

- *More Trivial MVD's.* If all the attributes of relation  $R$  are

$$\{A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m\}$$

then  $A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$  holds in  $R$ .

To see why these additional trivial MVD's hold, notice that if we take two tuples that agree in  $A_1, A_2, \dots, A_n$  and swap their components in attributes  $B_1, B_2, \dots, B_m$ , we get the same two tuples back, although in the opposite order.

### 3.6.4 Fourth Normal Form

The redundancy that we found in Section 3.6.1 to be caused by MVD's can be eliminated if we use these dependencies for decomposition. In this section we shall introduce a new normal form, called “fourth normal form.” In this normal form, all nontrivial MVD's are eliminated, as are all FD's that violate BCNF. As a result, the decomposed relations have neither the redundancy from FD's that we discussed in Section 3.3.1 nor the redundancy from MVD's that we discussed in Section 3.6.1.

The “fourth normal form” condition is essentially the BCNF condition, but applied to MVD's instead of FD's. Formally:

- A relation  $R$  is in *fourth normal form* (4NF) if whenever

$$A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$$

is a nontrivial MVD,  $\{A_1, A_2, \dots, A_n\}$  is a superkey.

That is, if a relation is in 4NF, then every nontrivial MVD is really an FD with a superkey on the left. Note that the notions of keys and superkeys depend on FD's only; adding MVD's does not change the definition of "key."

**Example 3.32:** The relation of Fig. 3.10 violates the 4NF condition. For example,

$$\text{name} \rightarrow\!\!\! \rightarrow \text{street city}$$

is a nontrivial MVD, yet **name** by itself is not a superkey. In fact, the only key for this relation is all the attributes.  $\square$

Fourth normal form is truly a generalization of BCNF. Recall from Section 3.6.3 that every FD is also an MVD. Thus, every BCNF violation is also a 4NF violation. Put another way, every relation that is in 4NF is therefore in BCNF.

However, there are some relations that are in BCNF but not 4NF. Figure 3.10 is a good example. The only key for this relation is all five attributes, and there are no nontrivial FD's. Thus it is surely in BCNF. However, as we observed in Example 3.32, it is not in 4NF.

### 3.6.5 Decomposition into Fourth Normal Form

The 4NF decomposition algorithm is quite analogous to the BCNF decomposition algorithm.

**Algorithm 3.33:** Decomposition into Fourth Normal Form.

**INPUT:** A relation  $R_0$  with a set of functional and multivalued dependencies  $S_0$ .

**OUTPUT:** A decomposition of  $R_0$  into relations all of which are in 4NF. The decomposition has the lossless-join property.

**METHOD:** Do the following steps, with  $R = R_0$  and  $S = S_0$ :

1. Find a 4NF violation in  $R$ , say  $A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$ , where

$$\{A_1, A_2, \dots, A_n\}$$

is not a superkey. Note this MVD could be a true MVD in  $S$ , or it could be derived from the corresponding FD  $A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$  in  $S$ , since every FD is an MVD. If there is none, return;  $R$  by itself is a suitable decomposition.

2. If there is such a 4NF violation, break the schema for the relation  $R$  that has the 4NF violation into two schemas:

- (a)  $R_1$ , whose schema is  $A$ 's and the  $B$ 's.
  - (b)  $R_2$ , whose schema is the  $A$ 's and all attributes of  $R$  that are not among the  $A$ 's or  $B$ 's.
3. Find the FD's and MVD's that hold in  $R_1$  and  $R_2$  (Section 3.7 explains how to do this task in general, but often this “projection” of dependencies is straightforward). Recursively decompose  $R_1$  and  $R_2$  with respect to their projected dependencies.

□

**Example 3.34:** Let us continue Example 3.32. We observed that

$$\text{name} \rightarrow\!\!\! \rightarrow \text{street city}$$

was a 4NF violation. The decomposition rule above tells us to replace the five-attribute schema by one schema that has only the three attributes in the above MVD and another schema that consists of the left side, `name`, plus the attributes that do not appear in the MVD. These attributes are `title` and `year`, so the following two schemas

$$\begin{aligned} &\{\text{name, street, city}\} \\ &\{\text{name, title, year}\} \end{aligned}$$

are the result of the decomposition. In each schema there are no nontrivial multivalued (or functional) dependencies, so they are in 4NF. Note that in the relation with schema `{name, street, city}`, the MVD:

$$\text{name} \rightarrow\!\!\! \rightarrow \text{street city}$$

is trivial since it involves all attributes. Likewise, in the relation with schema `{name, title, year}`, the MVD:

$$\text{name} \rightarrow\!\!\! \rightarrow \text{title year}$$

is trivial. Should one or both schemas of the decomposition not be in 4NF, we would have had to decompose the non-4NF schema(s). □

As for the BCNF decomposition, each decomposition step leaves us with schemas that have strictly fewer attributes than we started with, so eventually we get to schemas that need not be decomposed further; that is, they are in 4NF. Moreover, the argument justifying the decomposition that we gave in Section 3.4.1 carries over to MVD's as well. When we decompose a relation because of an MVD  $A_1 A_2 \cdots A_n \rightarrow\!\!\! \rightarrow B_1 B_2 \cdots B_m$ , this dependency is enough to justify the claim that we can reconstruct the original relation from the relations of the decomposition.

We shall, in Section 3.7, give an algorithm by which we can verify that the MVD used to justify a 4NF decomposition also proves that the decomposition has a lossless join. Also in that section, we shall show how it is possible, although time-consuming, to perform the projection of MVD's onto the decomposed relations. This projection is required if we are to decide whether or not further decomposition is necessary.

### 3.6.6 Relationships Among Normal Forms

As we have mentioned, 4NF implies BCNF, which in turn implies 3NF. Thus, the sets of relation schemas (including dependencies) satisfying the three normal forms are related as in Fig. 3.12. That is, if a relation with certain dependencies is in 4NF, it is also in BCNF and 3NF. Also, if a relation with certain dependencies is in BCNF, then it is in 3NF.

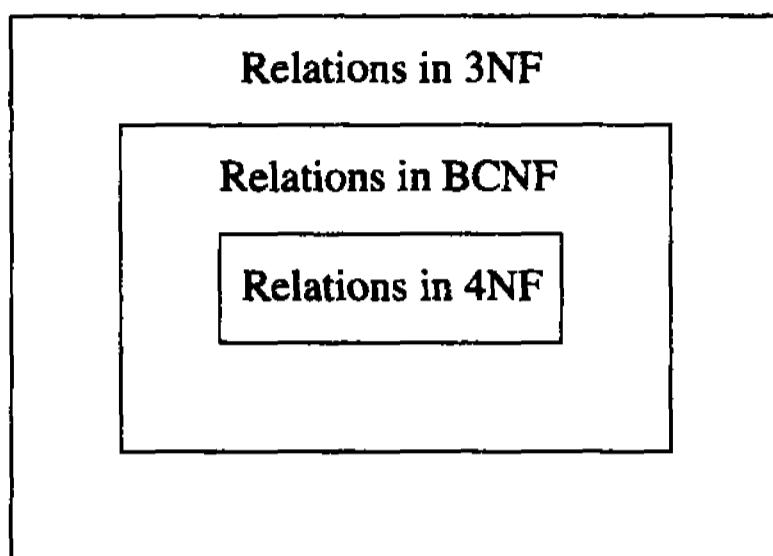


Figure 3.12: 4NF implies BCNF implies 3NF

Another way to compare the normal forms is by the guarantees they make about the set of relations that result from a decomposition into that normal form. These observations are summarized in the table of Fig. 3.13. That is, BCNF (and therefore 4NF) eliminates the redundancy and other anomalies that are caused by FD's, while only 4NF eliminates the additional redundancy that is caused by the presence of MVD's that are not FD's. Often, 3NF is enough to eliminate this redundancy, but there are examples where it is not. BCNF does not guarantee preservation of FD's, and none of the normal forms guarantee preservation of MVD's, although in typical cases the dependencies are preserved.

Property	3NF	BCNF	4NF
Eliminates redundancy due to FD's	No	Yes	Yes
Eliminates redundancy due to MVD's	No	No	Yes
Preserves FD's	Yes	No	No
Preserves MVD's	No	No	No

Figure 3.13: Properties of normal forms and their decompositions

### 3.6.7 Exercises for Section 3.6

**Exercise 3.6.1:** Suppose we have a relation  $R(A, B, C)$  with an MVD  $A \rightarrow\!\!\!-\> C$ .

B. If we know that the tuples  $(a, b_1, c_1)$ ,  $(a, b_2, c_2)$ , and  $(a, b_3, c_3)$  are in the current instance of  $R$ , what other tuples do we know must also be in  $R$ ?

**Exercise 3.6.2:** Suppose we have a relation in which we want to record for each person their name, Social Security number, and birthdate. Also, for each child of the person, the name, Social Security number, and birthdate of the child, and for each automobile the person owns, its serial number and make. To be more precise, this relation has all tuples

$$(n, s, b, cn, cs, cb, as, am)$$

such that

1.  $n$  is the name of the person with Social Security number  $s$ .
2.  $b$  is  $n$ 's birthdate.
3.  $cn$  is the name of one of  $n$ 's children.
4.  $cs$  is  $cn$ 's Social Security number.
5.  $cb$  is  $cn$ 's birthdate.
6.  $as$  is the serial number of one of  $n$ 's automobiles.
7.  $am$  is the make of the automobile with serial number  $as$ .

For this relation:

- a) Tell the functional and multivalued dependencies we would expect to hold.
- b) Suggest a decomposition of the relation into 4NF.

**Exercise 3.6.3:** For each of the following relation schemas and dependencies

- a)  $R(A, B, C, D)$  with MVD's  $A \rightarrow B$  and  $A \rightarrow C$ .
- b)  $R(A, B, C, D)$  with MVD's  $A \rightarrow B$  and  $B \rightarrow CD$ .
- c)  $R(A, B, C, D)$  with MVD  $AB \rightarrow C$  and FD  $B \rightarrow D$ .
- d)  $R(A, B, C, D, E)$  with MVD's  $A \rightarrow B$  and  $AB \rightarrow C$  and FD's  $A \rightarrow D$  and  $AB \rightarrow E$ .

do the following:

- i) Find all the 4NF violations.
- ii) Decompose the relations into a collection of relation schemas in 4NF.

**Exercise 3.6.4:** Give informal arguments why we would not expect any of the five attributes in Example 3.28 to be functionally determined by the other four.

## 3.7 An Algorithm for Discovering MVD's

Reasoning about MVD's, or combinations of MVD's and FD's, is rather more difficult than reasoning about FD's alone. For FD's, we have Algorithm 3.7 to decide whether or not an FD follows from some given FD's. In this section, we shall first show that the closure algorithm is really the same as the chase algorithm we studied in Section 3.4.2. The ideas behind the chase can be extended to incorporate MVD's as well as FD's. Once we have that tool in place, we can solve all the problems we need to solve about MVD's and FD's, such as finding whether an MVD follows from given dependencies or projecting MVD's and FD's onto the relations of a decomposition.

### 3.7.1 The Closure and the Chase

In Section 3.2.4 we saw how to take a set of attributes  $X$  and compute its closure  $X^+$  of all attributes that functionally depend on  $X$ . In that manner, we can test whether an FD  $X \rightarrow Y$  follows from a given set of FD's  $F$ , by closing  $X$  with respect to  $F$  and seeing whether  $Y \subseteq X^+$ . We could see the closure as a variant of the chase, in which the starting tableau and the goal condition are different from what we used in Section 3.4.2.

Suppose we start with a tableau that consists of two rows. These rows agree in the attributes of  $X$  and disagree in all other attributes. If we apply the FD's in  $F$  to chase this tableau, we shall equate the symbols in exactly those columns that are in  $X^+ - X$ . Thus, a chase-based test for whether  $X \rightarrow Y$  follows from  $F$  can be summarized as:

1. Start with a tableau having two rows that agree only on  $X$ .
2. Chase the tableau using the FD's of  $F$ .
3. If the final tableau agrees in all columns of  $Y$ , then  $X \rightarrow Y$  holds; otherwise it does not.

**Example 3.35:** Let us repeat Example 3.8, where we had a relation

$$R(A, B, C, D, E, F)$$

with FD's  $AB \rightarrow C$ ,  $BC \rightarrow AD$ ,  $D \rightarrow E$ , and  $CF \rightarrow B$ . We want to test whether  $AB \rightarrow D$  holds. Start with the tableau:

$A$	$B$	$C$	$D$	$E$	$F$
$a$	$b$	$c_1$	$d_1$	$e_1$	$f_1$
$a$	$b$	$c_2$	$d_2$	$e_2$	$f_2$

We can apply  $AB \rightarrow C$  to infer  $c_1 = c_2$ ; say both become  $c_1$ . The resulting tableau is:

A	B	C	D	E	F
a	b	$c_1$	$d_1$	$e_1$	$f_1$
a	b	$c_1$	$d_2$	$e_2$	$f_2$

Next, apply  $BC \rightarrow AD$  to infer that  $d_1 = d_2$ , and apply  $D \rightarrow E$  to infer  $e_1 = e_2$ . At this point, the tableau is:

A	B	C	D	E	F
a	b	$c_1$	$d_1$	$e_1$	$f_1$
a	b	$c_1$	$d_1$	$e_1$	$f_2$

and we can go no further. Since the two tuples now agree in the  $D$  column, we know that  $AB \rightarrow D$  does follow from the given FD's.  $\square$

### 3.7.2 Extending the Chase to MVD's

The method of inferring an FD using the chase can be applied to infer MVD's as well. When we try to infer an FD, we are asking whether we can conclude that two possibly unequal values must indeed be the same. When we apply an FD  $X \rightarrow Y$ , we search for pairs of rows in the tableau that agree on all the columns of  $X$ , and we force the symbols in each column of  $Y$  to be equal.

However, MVD's do not tell us to conclude symbols are equal. Rather,  $X \rightarrow\! \rightarrow Y$  tells us that if we find two rows of the tableau that agree in  $X$ , then we can form two new tuples by swapping all their components in the attributes of  $Y$ ; the resulting two tuples must also be in the relation, and therefore in the tableau. Likewise, if we want to infer some MVD  $X \rightarrow\! \rightarrow Y$  from given FD's and MVD's, we start with a tableau consisting of two tuples that agree in  $X$  and disagree in all attributes not in the set  $X$ . We apply the given FD's to equate symbols, and we apply the given MVD's to swap the values in certain attributes between two existing rows of the tableau in order to add new rows to the tableau. If we ever discover that one of the original tuples, with its components for  $Y$  replaced by those of the other original tuple, is in the tableau, then we have inferred the MVD.

There is a point of caution to be observed in this more complex chase process. Since symbols may get equated and replaced by other symbols, we may not recognize that we have created one of the desired tuples, because some of the original symbols may be replaced by others. The simplest way to avoid a problem is to define the target tuple initially, and never change its symbols. That is, let the target row be one with an unsubscripted letter in each component. Let the two initial rows of the tableau for the test of  $X \rightarrow\! \rightarrow Y$  have the unsubscripted letters in  $X$ . Let the first row also have unsubscripted letters in  $Y$ , and let the second row have the unsubscripted letters in all attributes not in  $X$  or  $Y$ . Fill in the other positions of the two rows with new symbols that each occur only once. When we equate subscripted and unsubscripted symbols, always replace a subscripted one by the unsubscripted one, as we did in Section 3.4.2. Then, when applying the chase, we have only to ask whether the all-unsubscripted-letters row ever appears in the tableau.

**Example 3.36:** Suppose we have a relation  $R(A, B, C, D)$  with given dependencies  $A \rightarrow B$  and  $B \rightarrow\rightarrow C$ . We wish to prove that  $A \rightarrow\rightarrow C$  holds in  $R$ . Start with the two-row tableau that represents  $A \rightarrow\rightarrow C$ :

A	B	C	D
$a$	$b_1$	$c$	$d_1$
$a$	$b$	$c_2$	$d$

Notice that our target row is  $(a, b, c, d)$ . Both rows of the tableau have the unsubscripted letter in the column for  $A$ . The first row has the unsubscripted letter in  $C$ , and the second row has unsubscripted letters in the remaining columns.

We first apply the FD  $A \rightarrow B$  to infer that  $b = b_1$ . We must therefore replace the subscripted  $b_1$  by the unsubscripted  $b$ . The tableau becomes:

A	B	C	D
$a$	$b$	$c$	$d_1$
$a$	$b$	$c_2$	$d$

Next, we apply the MVD  $B \rightarrow\rightarrow C$ , since the two rows now agree in the  $B$  column. We swap the  $C$  columns to get two more rows which we add to the tableau, which becomes:

A	B	C	D
$a$	$b$	$c$	$d_1$
$a$	$b$	$c_2$	$d$
$a$	$b$	$c_2$	$d_1$
$a$	$b$	$c$	$d$

We have now a row with all unsubscripted symbols, which proves that  $A \rightarrow\rightarrow C$  holds in relation  $R$ . Notice how the tableau manipulations really give a proof that  $A \rightarrow\rightarrow C$  holds. This proof is: “Given two tuples of  $R$  that agree in  $A$ , they must also agree in  $B$  because  $A \rightarrow B$ . Since they agree in  $B$ , we can swap their  $C$  components by  $B \rightarrow\rightarrow C$ , and the resulting tuples will be in  $R$ . Thus, if two tuples of  $R$  agree in  $A$ , the tuples that result when we swap their  $C$ ’s are also in  $R$ ; i.e.,  $A \rightarrow\rightarrow C$ .”  $\square$

**Example 3.37:** There is a surprising rule for FD’s and MVD’s that says whenever there is an MVD  $X \rightarrow\rightarrow Y$ , and any FD whose right side is a (not necessarily proper) subset of  $Y$ , say  $Z$ , then  $X \rightarrow Z$ . We shall use the chase process to prove a simple example of this rule. Let us be given relation  $R(A, B, C, D)$  with MVD  $A \rightarrow\rightarrow BC$  and FD  $D \rightarrow C$ . We claim that  $A \rightarrow C$ .

Since we are trying to prove an FD, we don’t have to worry about a target tuple of unsubscripted letters. We can start with any two tuples that agree in  $A$  and disagree in every other column. such as:

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>a</i>	$b_1$	$c_1$	$d_1$
<i>a</i>	$b_2$	$c_2$	$d_2$

Our goal is to prove that  $c_1 = c_2$ .

The only thing we can do to start is to apply the MVD  $A \rightarrow\!\!\! \rightarrow BC$ , since the two rows agree on  $A$ , but no other columns. When we swap the  $B$  and  $C$  columns of these two rows, we get two new rows to add:

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>a</i>	$b_1$	$c_1$	$d_1$
<i>a</i>	$b_2$	$c_2$	$d_2$
<i>a</i>	$b_2$	$c_2$	$d_1$
<i>a</i>	$b_1$	$c_1$	$d_2$

Now, we have pairs of rows that agree in  $D$ , so we can apply the FD  $D \rightarrow C$ . For instance, the first and third rows have the same  $D$ -value  $d_1$ , so we can apply the FD and conclude  $c_1 = c_2$ . That is our goal, so we have proved  $A \rightarrow C$ . The new tableau is:

<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>a</i>	$b_1$	$c_1$	$d_1$
<i>a</i>	$b_2$	$c_1$	$d_2$
<i>a</i>	$b_2$	$c_1$	$d_1$
<i>a</i>	$b_1$	$c_1$	$d_2$

It happens that no further changes are possible, using the given dependencies. However, that doesn't matter, since we already proved what we need.  $\square$

### 3.7.3 Why the Chase Works for MVD's

The arguments are essentially the same as we have given before. Each step of the chase, whether it equates symbols or generates new rows, is a true observation about tuples of the given relation  $R$  that is justified by the FD or MVD that we apply in that step. Thus, a positive conclusion of the chase is always a proof that the concluded FD or MVD holds in  $R$ .

When the chase ends in failure — the goal row (for an MVD) or the desired equality of symbols (for an FD) is not produced — then the final tableau is a counterexample. It satisfies the given dependencies, or else we would not be finished making changes. However, it does not satisfy the dependency we were trying to prove.

There is one other issue that did not come up when we performed the chase using only FD's. Since the chase with MVD's adds rows to the tableau, how do we know we ever terminate the chase? Could we keep adding rows forever, never reaching our goal, but not sure that after a few more steps we would achieve that goal? Fortunately, that cannot happen. The reason is that we

never create any new symbols. We start out with at most two symbols in each of  $k$  columns, and all rows we create will have one of these two symbols in its component for that column. Thus, we cannot ever have more than  $2^k$  rows in our tableau, if  $k$  is the number of columns. The chase with MVD's can take exponential time, but it cannot run forever.

### 3.7.4 Projecting MVD's

Recall that our reason for wanting to infer MVD's was to perform a cascade of decompositions leading to 4NF relations. To do that task, we need to be able to project the given dependencies onto the schemas of the two relations that we get in the first step of the decomposition. Only then can we know whether they are in 4NF or need to be decomposed further.

In the worst case, we have to test every possible FD and MVD for each of the decomposed relations. The chase test is applied on the full set of attributes of the original relation. However, the goal for an MVD is to produce a row of the tableau that has unsubscripted letters in all the attributes of one of the relations of the decomposition; that row may have any letters in the other attributes. The goal for an FD is the same: equality of the symbols in a given column.

**Example 3.38:** Suppose we have a relation  $R(A, B, C, D, E)$  that we decompose, and let one of the relations of the decomposition be  $S(A, B, C)$ . Suppose that the MVD  $A \rightarrow CD$  holds in  $R$ . Does this MVD imply any dependency in  $S$ ? We claim that  $A \rightarrow C$  holds in  $S$ , as does  $A \rightarrow B$  (by the complementation rule). Let us verify that  $A \rightarrow C$  holds in  $S$ . We start with the tableau:

$A$	$B$	$C$	$D$	$E$
$a$	$b_1$	$c$	$d_1$	$e_1$
$a$	$b$	$c_2$	$d$	$e$

Use the MVD of  $R$ ,  $A \rightarrow CD$  to swap the  $C$  and  $D$  components of these two rows to get two new rows:

$A$	$B$	$C$	$D$	$E$
$a$	$b_1$	$c$	$d_1$	$e_1$
$a$	$b$	$c_2$	$d$	$e$
$a$	$b_1$	$c_2$	$d$	$e_1$
$a$	$b$	$c$	$d_1$	$e$

Notice that the last row has unsubscripted symbols in all the attributes of  $S$ , that is,  $A$ ,  $B$ , and  $C$ . That is enough to conclude that  $A \rightarrow C$  holds in  $S$ .  $\square$

Often, our search for FD's and MVD's in the projected relations does not have to be completely exhaustive. Here are some simplifications.

1. It is surely not necessary to check the trivial FD's and MVD's.
2. For FD's, we can restrict ourselves to looking for FD's with a singleton right side, because of the combining rule for FD's.
3. An FD or MVD whose left side does not contain the left side of any given dependency surely cannot hold, since there is no way for its chase test to get started. That is, the two rows with which you start the test are unchanged by the given dependencies.

### 3.7.5 Exercises for Section 3.7

**Exercise 3.7.1:** Use the chase test to tell whether each of the following dependencies hold in a relation  $R(A, B, C, D, E)$  with the dependencies  $A \rightarrow\!\!> BC$ ,  $B \rightarrow D$ , and  $C \rightarrow E$ .

- a)  $A \rightarrow D$ .
- b)  $A \rightarrow\!\!> D$ .
- c)  $A \rightarrow E$ .
- d)  $A \rightarrow\!\!> E$ .

! **Exercise 3.7.2:** If we project the relation  $R$  of Exercise 3.7.1 onto  $S(A, C, E)$ , what nontrivial FD's and MVD's hold in  $S$ ?

! **Exercise 3.7.3:** Show the following rules for MVD's. In each case, you can set up the proof as a chase test, but you must think a little more generally than in the examples, since the set of attributes are arbitrary sets  $X$ ,  $Y$ ,  $Z$ , and the other unnamed attributes of the relation in which these dependencies hold.

- a) The *Union Rule*. If  $X$ ,  $Y$ , and  $Z$  are sets of attributes,  $X \rightarrow Y$ , and  $X \rightarrow Z$ , then  $X \rightarrow (Y \cup Z)$ .
- b) The *Intersection Rule*. If  $X$ ,  $Y$ , and  $Z$  are sets of attributes,  $X \rightarrow Y$ , and  $X \rightarrow Z$ , then  $X \rightarrow (Y \cap Z)$ .
- c) The *Difference Rule*. If  $X$ ,  $Y$ , and  $Z$  are sets of attributes,  $X \rightarrow Y$ , and  $X \rightarrow Z$ , then  $X \rightarrow (Y - Z)$ .
- d) *Removing attributes shared by left and right side*. If  $X \rightarrow Y$  holds, then  $X \rightarrow (Y - X)$  holds.

! **Exercise 3.7.4:** Give counterexample relations to show why the following rules for MVD's do *not* hold. *Hint:* apply the chase test and see what happens.

- a) If  $A \rightarrow\!\!> BC$ , then  $A \rightarrow B$ .
- b) If  $A \rightarrow B$ , then  $A \rightarrow\!\!> B$ .
- c) If  $AB \rightarrow\!\!> C$ , then  $A \rightarrow C$ .

## 3.8 Summary of Chapter 3

- ◆ *Functional Dependencies*: A functional dependency is a statement that two tuples of a relation that agree on some particular set of attributes must also agree on some other particular set of attributes.
- ◆ *Keys of a Relation*: A superkey for a relation is a set of attributes that functionally determines all the attributes of the relation. A key is a superkey, no proper subset of which is also a superkey.
- ◆ *Reasoning About Functional Dependencies*: There are many rules that let us infer that one FD  $X \rightarrow A$  holds in any relation instance that satisfies some other given set of FD's. To verify that  $X \rightarrow A$  holds, compute the closure of  $X$ , using the given FD's to expand  $X$  until it includes  $A$ .
- ◆ *Minimal Basis for a set of FD's*: For any set of FD's, there is at least one minimal basis, which is a set of FD's equivalent to the original (each set implies the other set), with singleton right sides, no FD that can be eliminated while preserving equivalence, and no attribute in a left side that can be eliminated while preserving equivalence.
- ◆ *Boyce-Codd Normal Form*: A relation is in BCNF if the only nontrivial FD's say that some superkey functionally determines one or more of the other attributes. A major benefit of BCNF is that it eliminates redundancy caused by the existence of FD's.
- ◆ *Lossless-Join Decomposition*: A useful property of a decomposition is that the original relation can be recovered exactly by taking the natural join of the relations in the decomposition. Any decomposition gives us back at least the tuples with which we start, but a carelessly chosen decomposition can give tuples in the join that were not in the original relation.
- ◆ *Dependency-Preserving Decomposition*: Another desirable property of a decomposition is that we can check all the functional dependencies that hold in the original relation by checking FD's in the decomposed relations.
- ◆ *Third Normal Form*: Sometimes decomposition into BCNF can lose the dependency-preservation property. A relaxed form of BCNF, called 3NF, allows an FD  $X \rightarrow A$  even if  $X$  is not a superkey, provided  $A$  is a member of some key. 3NF does not guarantee to eliminate all redundancy due to FD's, but often does so.
- ◆ *The Chase*: We can test whether a decomposition has the lossless-join property by setting up a tableau — a set of rows that represent tuples of the original relation. We chase a tableau by applying the given functional dependencies to infer that certain pairs of symbols must be the same. The decomposition is lossless with respect to a given set of FD's if and only if the chase leads to a row identical to the tuple whose membership in the join of the projected relations we assumed.

- ◆ *Synthesis Algorithm for 3NF*: If we take a minimal basis for a given set of FD's, turn each of these FD's into a relation, and add a key for the relation, if necessary, the result is a decomposition into 3NF that has the lossless-join and dependency-preservation properties.
- ◆ *Multivalued Dependencies*: A multivalued dependency is a statement that two sets of attributes in a relation have sets of values that appear in all possible combinations.
- ◆ *Fourth Normal Form*: MVD's can also cause redundancy in a relation. 4NF is like BCNF, but also forbids nontrivial MVD's whose left side is not a superkey. It is possible to decompose a relation into 4NF without losing information.
- ◆ *Reasoning About MVD's*: We can infer MVD's and FD's from a given set of MVD's and FD's by a chase process. We start with a two-row tableau that represent the dependency we are trying to prove. FD's are applied by equating symbols, and MVD's are applied by adding rows to the tableau that have the appropriate components interchanged.

## 3.9 References for Chapter 3

Third normal form was described in [6]. This paper introduces the idea of functional dependencies, as well as the basic relational concept. Boyce-Codd normal form is in a later paper [7].

Multivalued dependencies and fourth normal form were defined by Fagin in [9]. However, the idea of multivalued dependencies also appears independently in [8] and [11].

Armstrong was the first to study rules for inferring FD's [2]. The rules for FD's that we have covered here (including what we call "Armstrong's axioms") and rules for inferring MVD's as well, come from [3].

The technique for testing an FD by computing the closure for a set of attributes is from [4], as is the fact that a minimal basis provides a 3NF decomposition. The fact that this decomposition provides the lossless-join and dependency-preservation properties is from [5].

The tableau test for the lossless-join property and the chase are from [1]. More information and the history of the idea is found in [10].

1. A. V. Aho, C. Beeri, and J. D. Ullman, "The theory of joins in relational databases," *ACM Transactions on Database Systems* 4:3, pp. 297-314, 1979.
2. W. W. Armstrong, "Dependency structures of database relationships," *Proceedings of the 1974 IFIP Congress*, pp. 580-583.

3. C. Beeri, R. Fagin, and J. H. Howard, "A complete axiomatization for functional and multivalued dependencies," *ACM SIGMOD International Conference on Management of Data*, pp. 47–61, 1977.
4. P. A. Bernstein, "Synthesizing third normal form relations from functional dependencies," *ACM Transactions on Database Systems* 1:4, pp. 277–298, 1976.
5. J. Biskup, U. Dayal, and P. A. Bernstein, "Synthesizing independent database schemas," *ACM SIGMOD International Conference on Management of Data*, pp. 143–152, 1979.
6. E. F. Codd, "A relational model for large shared data banks," *Comm. ACM* 13:6, pp. 377–387, 1970.
7. E. F. Codd, "Further normalization of the data base relational model," in *Database Systems* (R. Rustin, ed.), Prentice-Hall, Englewood Cliffs, NJ, 1972.
8. C. Delobel, "Normalization and hierarchical dependencies in the relational data model," *ACM Transactions on Database Systems* 3:3, pp. 201–222, 1978.
9. R. Fagin, "Multivalued dependencies and a new normal form for relational databases," *ACM Transactions on Database Systems* 2:3, pp. 262–278, 1977.
10. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.
11. C. Zaniolo and M. A. Melkanoff, "On the design of relational database schemata," *ACM Transactions on Database Systems* 6:1, pp. 1–47, 1981.



# Chapter 4

## High-Level Database Models

Let us consider the process whereby a new database, such as our movie database, is created. Figure 4.1 suggests the process. We begin with a design phase, in which we address and answer questions about what information will be stored, how information elements will be related to one another, what constraints such as keys or referential integrity may be assumed, and so on. This phase may last for a long time, while options are evaluated and opinions are reconciled. We show this phase in Fig. 4.1 as the conversion of ideas to a high-level design.

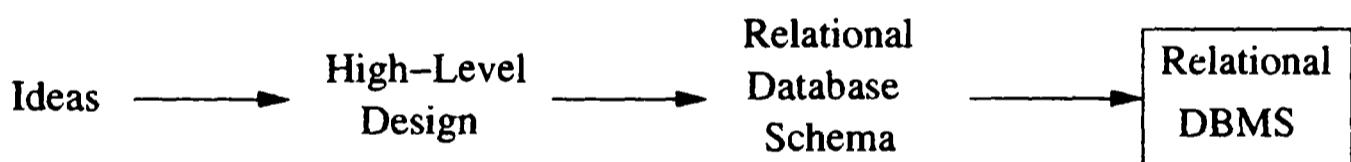


Figure 4.1: The database modeling and implementation process

Since the great majority of commercial database systems use the relational model, we might suppose that the design phase should use this model too. However, in practice it is often easier to start with a higher-level model and then convert the design to the relational model. The primary reason for doing so is that the relational model has only one concept — the relation — rather than several complementary concepts that more closely model real-world situations. Simplicity of concepts in the relational model is a great strength of the model, especially when it comes to efficient implementation of database operations. Yet that strength becomes a weakness when we do a preliminary design, which is why it often is helpful to begin by using a high-level design model.

There are several options for the notation in which the design is expressed. The first, and oldest, method is the “entity-relationship diagram,” and here is where we shall start in Section 4.1. A more recent trend is the use of UML (“Unified Modeling Language”), a notation that was originally designed for

describing object-oriented software projects, but which has been adapted to describe database schemas as well. We shall see this model in Section 4.7. Finally, in Section 4.9, we shall consider ODL (“Object Description Language”), which was created to describe databases as collections of classes and their objects.

The next phase shown in Fig. 4.1 is the conversion of our high-level design to a relational design. This phase occurs only when we are confident of the high-level design. Whichever of the high-level models we use, there is a fairly mechanical way of converting the high-level design into a relational database schema, which then runs on a conventional DBMS. Sections 4.5 and 4.6 discuss conversion of E/R diagrams to relational database schemas. Section 4.8 does the same for UML, and Section 4.10 serves for ODL.

## 4.1 The Entity/Relationship Model

In the *entity-relationship model* (or *E/R model*), the structure of data is represented graphically, as an “entity-relationship diagram,” using three principal element types:

1. Entity sets,
2. Attributes, and
3. Relationships.

We shall cover each in turn.

### 4.1.1 Entity Sets

An *entity* is an abstract object of some sort, and a collection of similar entities forms an *entity set*. An entity in some ways resembles an “object” in the sense of object-oriented programming. Likewise, an entity set bears some resemblance to a class of objects. However, the E/R model is a static concept, involving the structure of data and not the operations on data. Thus, one would not expect to find methods associated with an entity set as one would with a class.

**Example 4.1:** Let us consider the design of our running movie-database example. Each movie is an entity, and the set of all movies constitutes an entity set. Likewise, the stars are entities, and the set of stars is an entity set. A studio is another kind of entity, and the set of studios is a third entity set that will appear in our examples. □

### 4.1.2 Attributes

Entity sets have associated *attributes*, which are properties of the entities in that set. For instance, the entity set *Movies* might be given attributes such as *title* and *length*. It should not surprise you if the attributes for the entity

## E/R Model Variations

In some versions of the E/R model, the type of an attribute can be either:

1. A primitive type, as in the version presented here.
2. A “struct,” as in C, or tuple with a fixed number of primitive components.
3. A set of values of one type: either primitive or a “struct” type.

For example, the type of an attribute in such a model could be a set of pairs, each pair consisting of an integer and a string.

set *Movies* resemble the attributes of the relation **Movies** in our example. It is common for entity sets to be implemented as relations, although not every relation in our final relational design will come from an entity set.

In our version of the E/R model, we shall assume that attributes are of primitive types, such as strings, integers, or reals. There are other variations of this model in which attributes can have some limited structure; see the box on “E/R Model Variations.”

### 4.1.3 Relationships

*Relationships* are connections among two or more entity sets. For instance, if *Movies* and *Stars* are two entity sets, we could have a relationship *Stars-in* that connects movies and stars. The intent is that a movie entity  $m$  is related to a star entity  $s$  by the relationship *Stars-in* if  $s$  appears in movie  $m$ . While binary relationships, those between two entity sets, are by far the most common type of relationship, the E/R model allows relationships to involve any number of entity sets. We shall defer discussion of these multiway relationships until Section 4.1.7.

### 4.1.4 Entity-Relationship Diagrams

An *E/R diagram* is a graph representing entity sets, attributes, and relationships. Elements of each of these kinds are represented by nodes of the graph, and we use a special shape of node to indicate the kind, as follows:

- Entity sets are represented by rectangles.
- Attributes are represented by ovals.
- Relationships are represented by diamonds.

Edges connect an entity set to its attributes and also connect a relationship to its entity sets.

**Example 4.2 :** In Fig. 4.2 is an E/R diagram that represents a simple database about movies. The entity sets are *Movies*, *Stars*, and *Studios*.

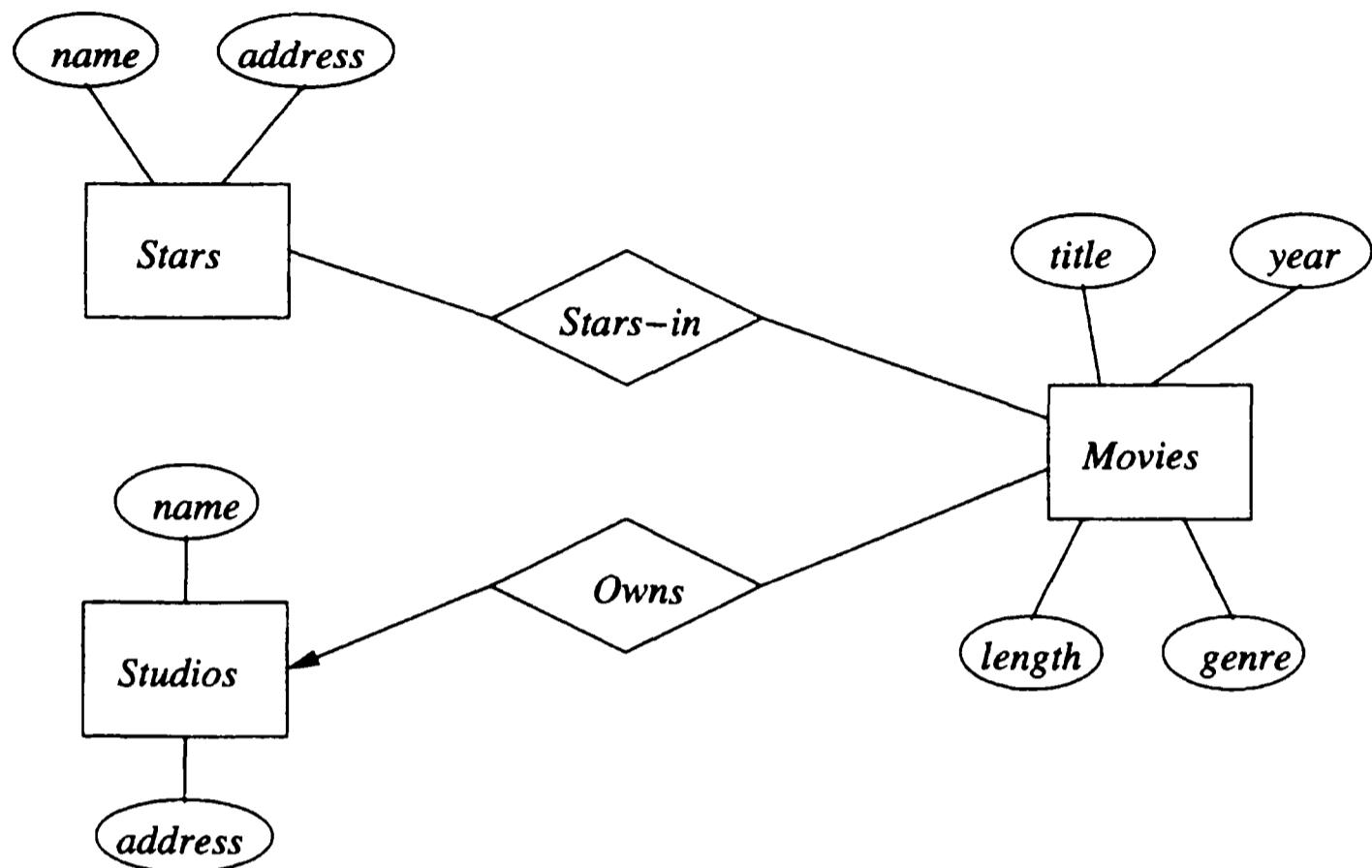


Figure 4.2: An entity-relationship diagram for the movie database

The *Movies* entity set has four of our usual attributes: *title*, *year*, *length*, and *genre*. The other two entity sets *Stars* and *Studios* happen to have the same two attributes: *name* and *address*, each with an obvious meaning. We also see two relationships in the diagram:

1. *Stars-in* is a relationship connecting each movie to the stars of that movie. This relationship consequently also connects stars to the movies in which they appeared.
2. *Owns* connects each movie to the studio that owns the movie. The arrow pointing to entity set *Studios* in Fig. 4.2 indicates that each movie is owned by at most one studio. We shall discuss uniqueness constraints such as this one in Section 4.1.6.

□

#### 4.1.5 Instances of an E/R Diagram

E/R diagrams are a notation for describing schemas of databases. We may imagine that a database described by an E/R diagram contains particular data, an “instance” of the database. Since the database is not implemented in the E/R model, only designed, the instance never exists in the sense that a relation’s

instances exist in a DBMS. However, it is often useful to visualize the database being designed as if it existed.

For each entity set, the database instance will have a particular finite set of entities. Each of these entities has particular values for each attribute. A relationship  $R$  that connects  $n$  entity sets  $E_1, E_2, \dots, E_n$  may be imagined to have an “instance” that consists of a finite set of tuples  $(e_1, e_2, \dots, e_n)$ , where each  $e_i$  is chosen from the entities that are in the current instance of entity set  $E_i$ . We regard each of these tuples as “connected” by relationship  $R$ .

This set of tuples is called the *relationship set* for  $R$ . It is often helpful to visualize a relationship set as a table or relation. However, the “tuples” of a relationship set are not really tuples of a relation, since their components are entities rather than primitive types such as strings or integers. The columns of the table are headed by the names of the entity sets involved in the relationship, and each list of connected entities occupies one row of the table. As we shall see, however, when we convert relationships to relations, the resulting relation is not the same as the relationship set.

**Example 4.3:** An instance of the *Stars-in* relationship could be visualized as a table with pairs such as:

Movies	Stars
Basic Instinct	Sharon Stone
Total Recall	Arnold Schwarzenegger
Total Recall	Sharon Stone

The members of the relationship set are the rows of the table. For instance, (Basic Instinct, Sharon Stone) is a tuple in the relationship set for the current instance of relationship *Stars-in*.  $\square$

#### 4.1.6 Multiplicity of Binary E/R Relationships

In general, a binary relationship can connect any member of one of its entity sets to any number of members of the other entity set. However, it is common for there to be a restriction on the “multiplicity” of a relationship. Suppose  $R$  is a relationship connecting entity sets  $E$  and  $F$ . Then:

- If each member of  $E$  can be connected by  $R$  to at most one member of  $F$ , then we say that  $R$  is *many-one* from  $E$  to  $F$ . Note that in a many-one relationship from  $E$  to  $F$ , each entity in  $F$  can be connected to many members of  $E$ . Similarly, if instead a member of  $F$  can be connected by  $R$  to at most one member of  $E$ , then we say  $R$  is many-one from  $F$  to  $E$  (or equivalently, one-many from  $E$  to  $F$ ).
- If  $R$  is both many-one from  $E$  to  $F$  and many-one from  $F$  to  $E$ , then we say that  $R$  is *one-one*. In a one-one relationship an entity of either entity set can be connected to at most one entity of the other set.

- If  $R$  is neither many-one from  $E$  to  $F$  or from  $F$  to  $E$ , then we say  $R$  is *many-many*.

As we mentioned in Example 4.2, arrows can be used to indicate the multiplicity of a relationship in an E/R diagram. If a relationship is many-one from entity set  $E$  to entity set  $F$ , then we place an arrow entering  $F$ . The arrow indicates that each entity in set  $E$  is related to at most one entity in set  $F$ . Unless there is also an arrow on the edge to  $E$ , an entity in  $F$  may be related to many entities in  $E$ .

**Example 4.4:** A one-one relationship between entity sets  $E$  and  $F$  is represented by arrows pointing to both  $E$  and  $F$ . For instance, Fig. 4.3 shows two entity sets, *Studios* and *Presidents*, and the relationship *Runs* between them (attributes are omitted). We assume that a president can run only one studio and a studio has only one president, so this relationship is one-one, as indicated by the two arrows, one entering each entity set.



Figure 4.3: A one-one relationship

Remember that the arrow means “at most one”; it does not guarantee existence of an entity of the set pointed to. Thus, in Fig. 4.3, we would expect that a “president” is surely associated with some studio; how could they be a “president” otherwise? However, a studio might not have a president at some particular time, so the arrow from *Runs* to *Presidents* truly means “at most one” and not “exactly one.” We shall discuss the distinction further in Section 4.3.3.

□

#### 4.1.7 Multiway Relationships

The E/R model makes it convenient to define relationships involving more than two entity sets. In practice, ternary (three-way) or higher-degree relationships are rare, but they occasionally are necessary to reflect the true state of affairs. A multiway relationship in an E/R diagram is represented by lines from the relationship diamond to each of the involved entity sets.

**Example 4.5:** In Fig. 4.4 is a relationship *Contracts* that involves a studio, a star, and a movie. This relationship represents that a studio has contracted with a particular star to act in a particular movie. In general, the value of an E/R relationship can be thought of as a relationship set of tuples whose components are the entities participating in the relationship, as we discussed in Section 4.1.5. Thus, relationship *Contracts* can be described by triples of the form (studio, star, movie).

## Implications Among Relationship Types

We should be aware that a many-one relationship is a special case of a many-many relationship, and a one-one relationship is a special case of a many-one relationship. Thus, any useful property of many-one relationships holds for one-one relationships too. For example, a data structure for representing many-one relationships will work for one-one relationships, although it might not be suitable for many-many relationships.

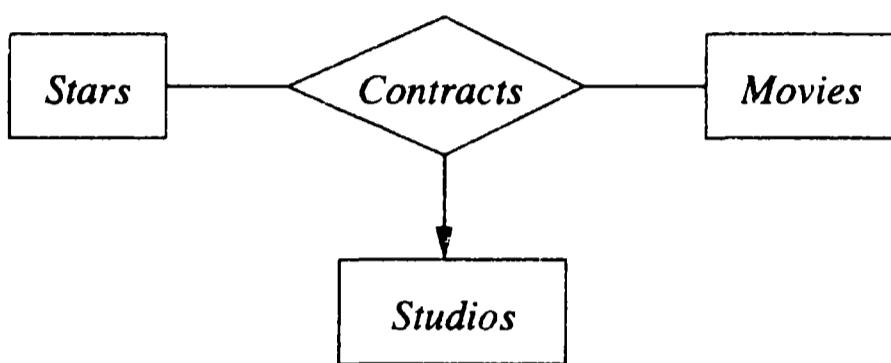


Figure 4.4: A three-way relationship

In multiway relationships, an arrow pointing to an entity set  $E$  means that if we select one entity from each of the other entity sets in the relationship, those entities are related to at most one entity in  $E$ . (Note that this rule generalizes the notation used for many-one, binary relationships.) Informally, we may think of a functional dependency with  $E$  on the right and all the other entity sets of the relationship on the left.

In Fig. 4.4 we have an arrow pointing to entity set *Studios*, indicating that for a particular star and movie, there is only one studio with which the star has contracted for that movie. However, there are no arrows pointing to entity sets *Stars* or *Movies*. A studio may contract with several stars for a movie, and a star may contract with one studio for more than one movie.  $\square$

### 4.1.8 Roles in Relationships

It is possible that one entity set appears two or more times in a single relationship. If so, we draw as many lines from the relationship to the entity set as the entity set appears in the relationship. Each line to the entity set represents a different *role* that the entity set plays in the relationship. We therefore label the edges between the entity set and relationship by names, which we call “roles.”

**Example 4.6:** In Fig. 4.5 is a relationship *Sequel-of* between the entity set *Movies* and itself. Each relationship is between two movies, one of which is the sequel of the other. To differentiate the two movies in a relationship, one line is labeled by the role *Original* and one by the role *Sequel*, indicating the

## Limits on Arrow Notation in Multiway Relationships

There are not enough choices of arrow or no-arrow on the lines attached to a relationship with three or more participants. Thus, we cannot describe every possible situation with arrows. For instance, in Fig. 4.4, the studio is really a function of the movie alone, not the star and movie jointly, since only one studio produces a movie. However, our notation does not distinguish this situation from the case of a three-way relationship where the entity set pointed to by the arrow is truly a function of both other entity sets. To handle all possible situations, we would have to give a set of functional dependencies involving the entity sets of the relationship.

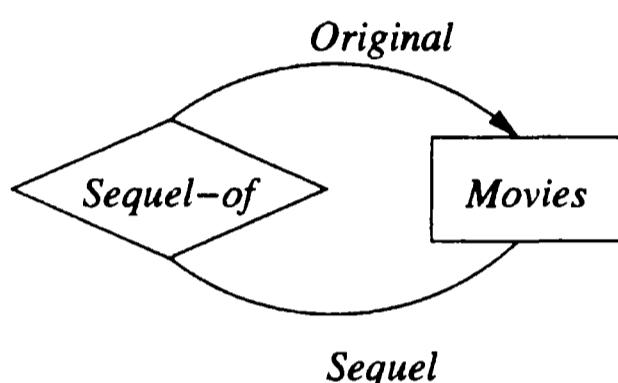


Figure 4.5: A relationship with roles

original movie and its sequel, respectively. We assume that a movie may have many sequels, but for each sequel there is only one original movie. Thus, the relationship is many-one from *Sequel* movies to *Original* movies, as indicated by the arrow in the E/R diagram of Fig. 4.5. □

**Example 4.7:** As a final example that includes both a multiway relationship and an entity set with multiple roles, in Fig. 4.6 is a more complex version of the *Contracts* relationship introduced earlier in Example 4.5. Now, relationship *Contracts* involves two studios, a star, and a movie. The intent is that one studio, having a certain star under contract (in general, not for a particular movie), may further contract with a second studio to allow that star to act in a particular movie. Thus, the relationship is described by 4-tuples of the form (studio1, studio2, star, movie), meaning that studio2 contracts with studio1 for the use of studio1's star by studio2 for the movie.

We see in Fig. 4.6 arrows pointing to *Studios* in both of its roles, as “owner” of the star and as producer of the movie. However, there are not arrows pointing to *Stars* or *Movies*. The rationale is as follows. Given a star, a movie, and a studio producing the movie, there can be only one studio that “owns” the star. (We assume a star is under contract to exactly one studio.) Similarly, only one studio produces a given movie, so given a star, a movie, and the star's studio, we can determine a unique producing studio. Note that in both

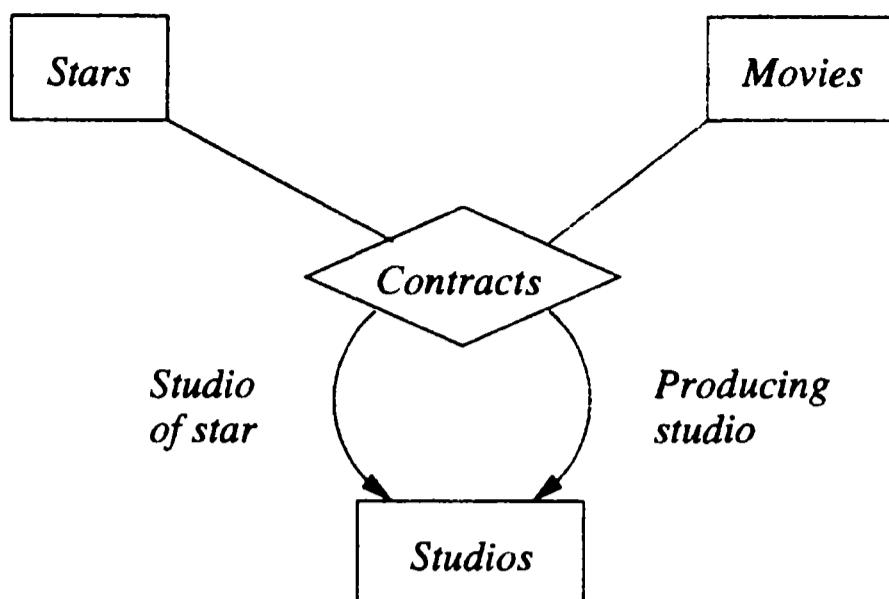


Figure 4.6: A four-way relationship

cases we actually needed only one of the other entities to determine the unique entity—for example, we need only know the movie to determine the unique producing studio—but this fact does not change the multiplicity specification for the multiway relationship.

There are no arrows pointing to *Stars* or *Movies*. Given a star, the star's studio, and a producing studio, there could be several different contracts allowing the star to act in several movies. Thus, the other three components in a relationship 4-tuple do not necessarily determine a unique movie. Similarly, a producing studio might contract with some other studio to use more than one of their stars in one movie. Thus, a star is not determined by the three other components of the relationship. □

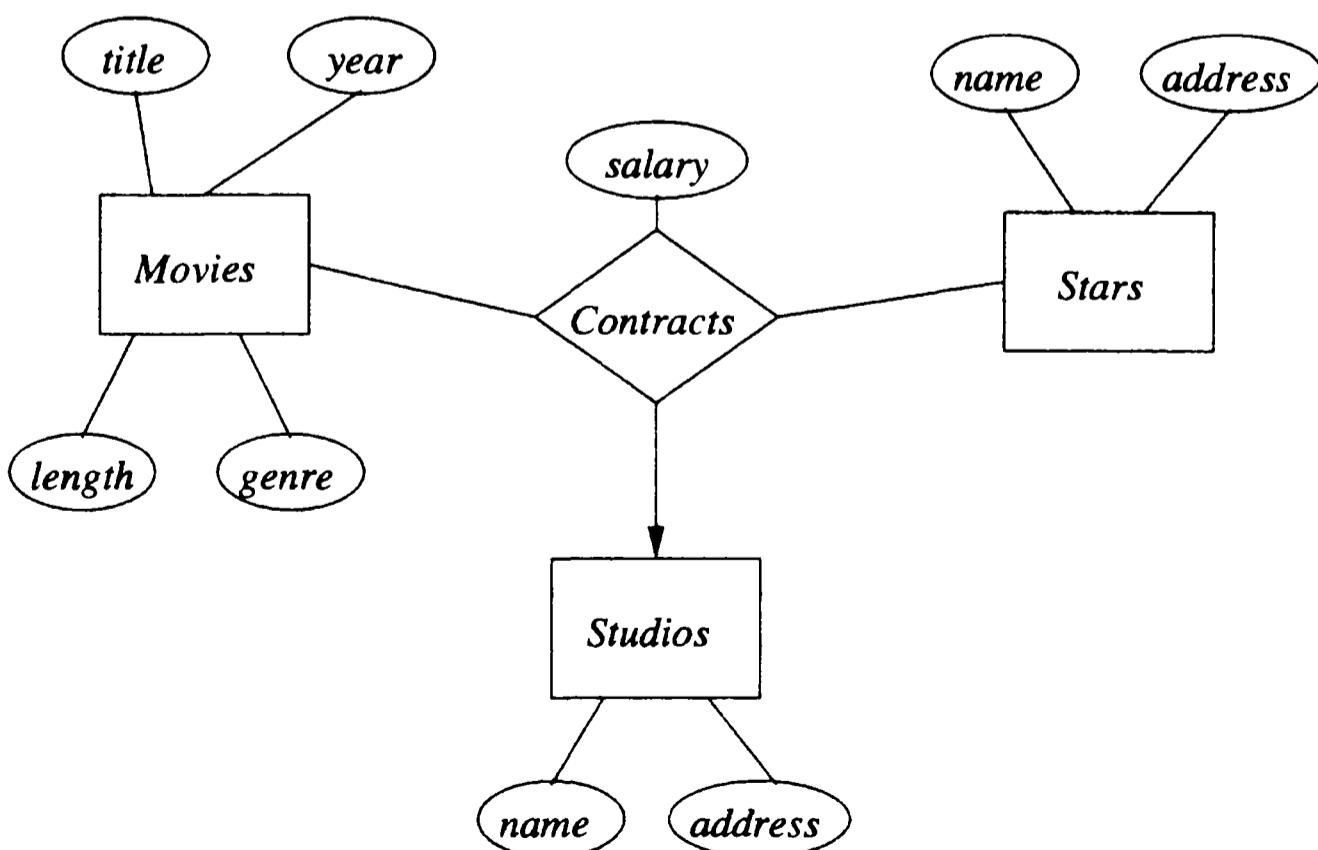


Figure 4.7: A relationship with an attribute

### 4.1.9 Attributes on Relationships

Sometimes it is convenient, or even essential, to associate attributes with a relationship, rather than with any one of the entity sets that the relationship connects. For example, consider the relationship of Fig. 4.4, which represents contracts between a star and studio for a movie.<sup>1</sup> We might wish to record the salary associated with this contract. However, we cannot associate it with the star; a star might get different salaries for different movies. Similarly, it does not make sense to associate the salary with a studio (they may pay different salaries to different stars) or with a movie (different stars in a movie may receive different salaries).

However, we can associate a unique salary with the (star, movie, studio) triple in the relationship set for the *Contracts* relationship. In Fig. 4.7 we see Fig. 4.4 fleshed out with attributes. The relationship has attribute *salary*, while the entity sets have the same attributes that we showed for them in Fig. 4.2.

In general, we may place one or more attributes on any relationship. The values of these attributes are functionally determined by the entire tuple in the relationship set for that relation. In some cases, the attributes can be determined by a subset of the entity sets involved in the relation, but presumably not by any single entity set (or it would make more sense to place the attribute on that entity set). For instance, in Fig. 4.7, the salary is really determined by the movie and star entities, since the studio entity is itself determined by the movie entity.

It is never necessary to place attributes on relationships. We can instead invent a new entity set, whose entities have the attributes ascribed to the relationship. If we then include this entity set in the relationship, we can omit the attributes on the relationship itself. However, attributes on a relationship are a useful convention, which we shall continue to use where appropriate.

**Example 4.8:** Let us revise the E/R diagram of Fig. 4.7, which has the salary attribute on the *Contracts* relationship. Instead, we create an entity set *Salaries*, with attribute *salary*. *Salaries* becomes the fourth entity set of relationship *Contracts*. The whole diagram is shown in Fig. 4.8.

Notice that there is an arrow into the *Salaries* entity set in Fig. 4.8. That arrow is appropriate, since we know that the salary is determined by all the other entity sets involved in the relationship. In general, when we do a conversion from attributes on a relationship to an additional entity set, we place an arrow into that entity set. □

### 4.1.10 Converting Multiway Relationships to Binary

There are some data models, such as UML (Section 4.7) and ODL (Section 4.9), that limit relationships to be binary. Thus, while the E/R model does not

---

<sup>1</sup>Here, we have reverted to the earlier notion of three-way contracts in Example 4.5, not the four-way relationship of Example 4.7.

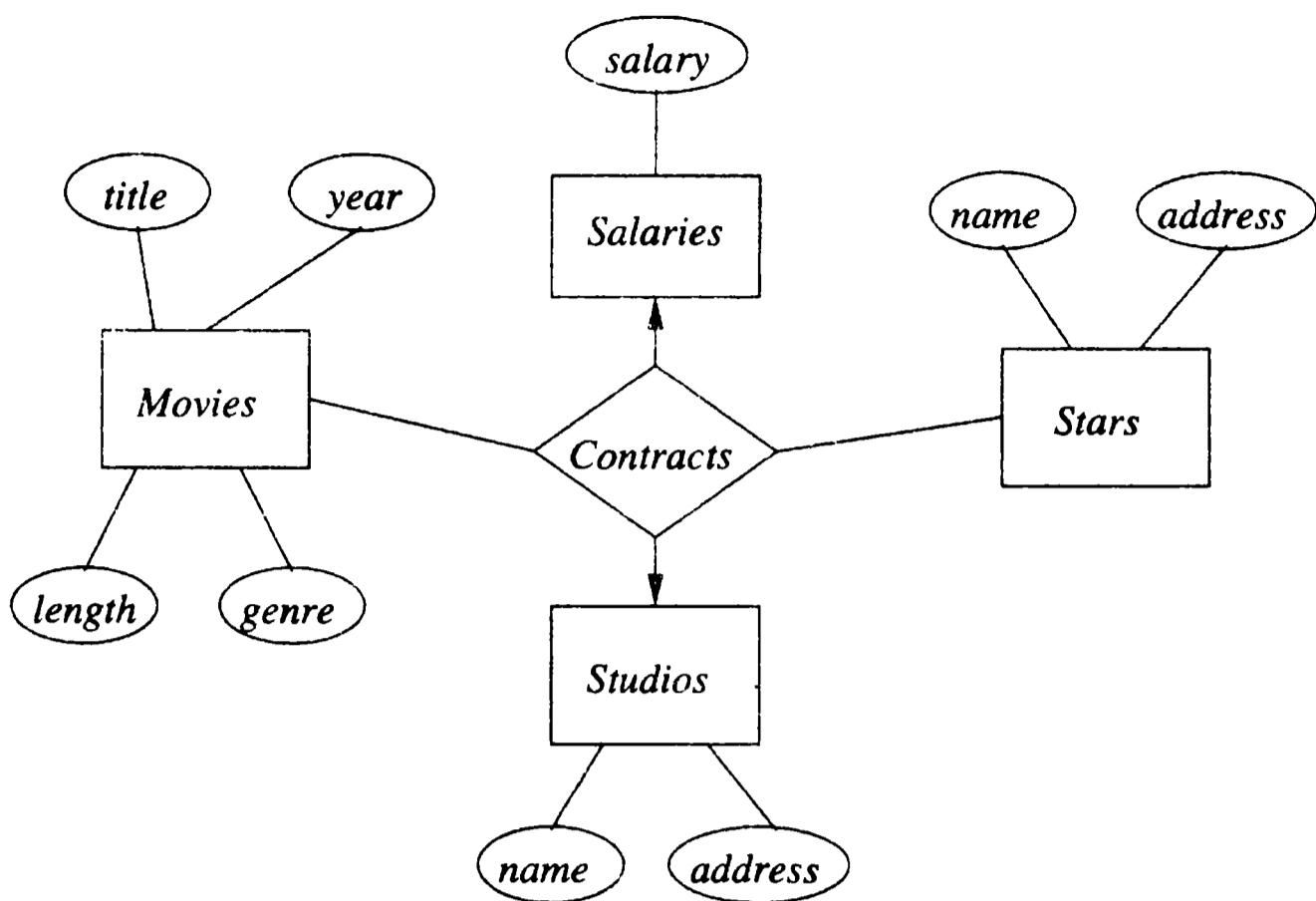


Figure 4.8: Moving the attribute to an entity set

require binary relationships, it is useful to observe that any relationship connecting more than two entity sets can be converted to a collection of binary, many-one relationships. To do so, introduce a new entity set whose entities we may think of as tuples of the relationship set for the multiway relationship. We call this entity set a *connecting* entity set. We then introduce many-one relationships from the connecting entity set to each of the entity sets that provide components of tuples in the original, multiway relationship. If an entity set plays more than one role, then it is the target of one relationship for each role.

**Example 4.9:** The four-way *Contracts* relationship in Fig. 4.6 can be replaced by an entity set that we may also call *Contracts*. As seen in Fig. 4.9, it participates in four relationships. If the relationship set for the relationship *Contracts* has a 4-tuple (studio1, studio2, star, movie) then the entity set *Contracts* has an entity *e*. This entity is linked by relationship *Star-of* to the entity *star* in entity set *Stars*. It is linked by relationship *Movie-of* to the entity *movie* in *Movies*. It is linked to entities *studio1* and *studio2* of *Studios* by relationships *Studio-of-star* and *Producing-studio*, respectively.

Note that we have assumed there are no attributes of entity set *Contracts*, although the other entity sets in Fig. 4.9 have unseen attributes. However, it is possible to add attributes, such as the date of signing, to entity set *Contracts*.  $\square$

#### 4.1.11 Subclasses in the E/R Model

Often, an entity set contains certain entities that have special properties not associated with all members of the set. If so, we find it useful to define certain

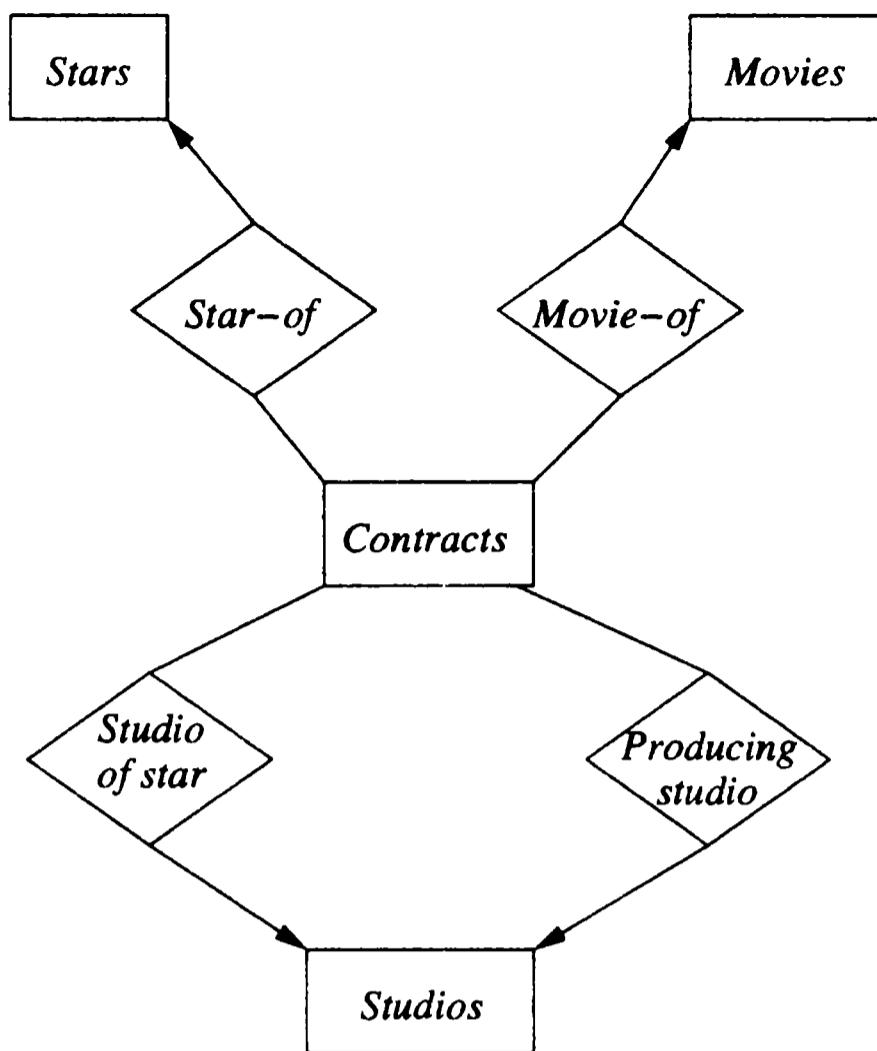


Figure 4.9: Replacing a multiway relationship by an entity set and binary relationships

special-case entity sets, or *subclasses*, each with its own special attributes and/or relationships. We connect an entity set to its subclasses using a relationship called *isa* (i.e., “an *A* is a *B*” expresses an “*isa*” relationship from entity set *A* to entity set *B*).

An *isa* relationship is a special kind of relationship, and to emphasize that it is unlike other relationships, we use a special notation: a triangle. One side of the triangle is attached to the subclass, and the opposite point is connected to the superclass. Every *isa* relationship is one-one, although we shall not draw the two arrows that are associated with other one-one relationships.

**Example 4.10:** Among the special kinds of movies we might store in our example database are cartoons and murder mysteries. For each of these special movie types, we could define a subclass of the entity set *Movies*. For instance, let us postulate two subclasses: *Cartoons* and *Murder-Mysteries*. A cartoon has, in addition to the attributes and relationships of *Movies*, an additional relationship called *Voices* that gives us a set of stars who speak, but do not appear in the movie. Movies that are not cartoons do not have such stars. Murder-mysteries have an additional attribute *weapon*. The connections among the three entity sets *Movies*, *Cartoons*, and *Murder-Mysteries* is shown in Fig. 4.10. □

While, in principle, a collection of entity sets connected by *isa* relationships could have any structure, we shall limit *isa*-structures to trees, in which there

## Parallel Relationships Can Be Different

Figure 4.9 illustrates a subtle point about relationships. There are two different relationships, *Studio-of-Star* and *Producing-Studio*, that each connect entity sets *Contracts* and *Studios*. We should not presume that these relationships therefore have the same relationship sets. In fact, in this case, it is unlikely that both relationships would ever relate the same contract to the same studios, since a studio would then be contracting with itself.

More generally, there is nothing wrong with an E/R diagram having several relationships that connect the same entity sets. In the database, the instances of these relationships will normally be different, reflecting the different meanings of the relationships.

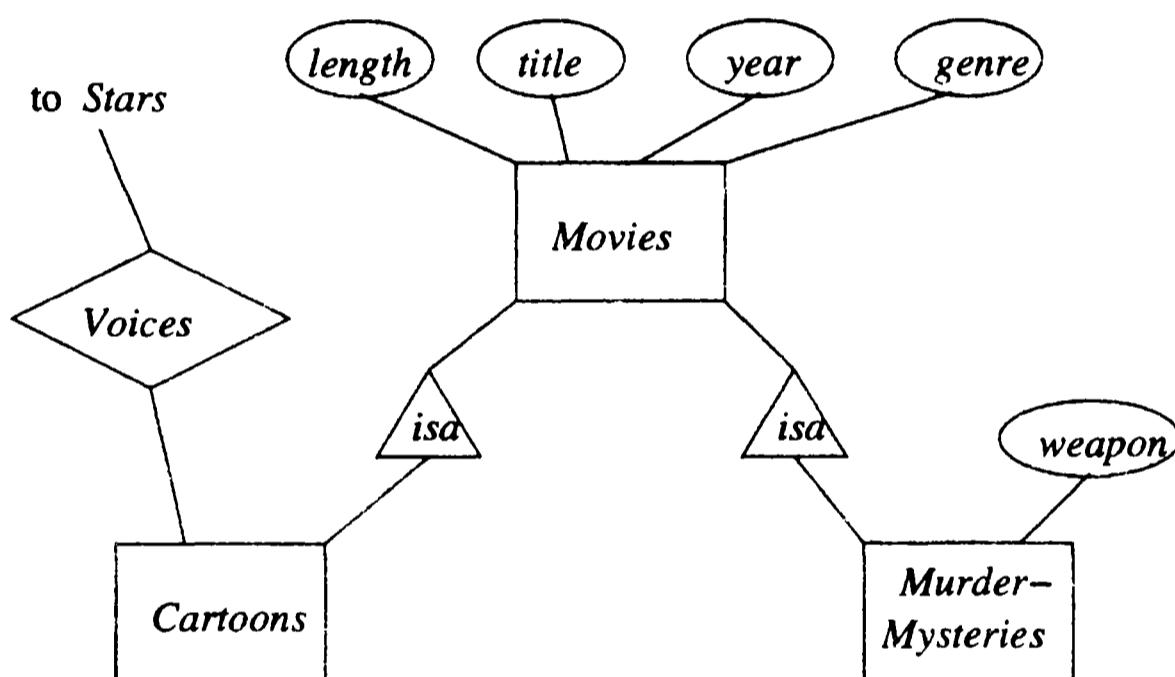


Figure 4.10: Isa relationships in an E/R diagram

is one *root* entity set (e.g., *Movies* in Fig. 4.10) that is the most general, with progressively more specialized entity sets extending below the root in a tree.

Suppose we have a tree of entity sets, connected by *isa* relationships. A single entity consists of *components* from one or more of these entity sets, as long as those components are in a subtree including the root. That is, if an entity *e* has a component *c* in entity set *E*, and the parent of *E* in the tree is *F*, then entity *e* also has a component *d* in *F*. Further, *c* and *d* must be paired in the relationship set for the *isa* relationship from *E* to *F*. The entity *e* has whatever attributes any of its components has, and it participates in whatever relationships any of its components participate in.

**Example 4.11:** The typical movie, being neither a cartoon nor a murder-mystery, will have a component only in the root entity set *Movies* in Fig. 4.10. These entities have only the four attributes of *Movies* (and the two relationships

## The E/R View of Subclasses

There is a significant resemblance between “isa” in the E/R model and subclasses in object-oriented languages. In a sense, “isa” relates a subclass to its superclass. However, there is also a fundamental difference between the conventional E/R view and the object-oriented approach: entities are allowed to have representatives in a tree of entity sets, while objects are assumed to exist in exactly one class or subclass.

The difference becomes apparent when we consider how the movie *Roger Rabbit* was handled in Example 4.11. In an object-oriented approach, we would need for this movie a fourth entity set, “cartoon-murder-mystery,” which inherited all the attributes and relationships of *Movies*, *Cartoons*, and *Murder-Mysteries*. However, in the E/R model, the effect of this fourth subclass is obtained by putting components of the movie *Roger Rabbit* in both the *Cartoons* and *Murder-Mysteries* entity sets.

of *Movies* — *Stars-in* and *Owns* — that are not shown in Fig. 4.10).

A cartoon that is not a murder-mystery will have two components, one in *Movies* and one in *Cartoons*. Its entity will therefore have not only the four attributes of *Movies*, but the relationship *Vocies*. Likewise, a murder-mystery will have two components for its entity, one in *Movies* and one in *Murder-Mysteries* and thus will have five attributes, including *weapon*.

Finally, a movie like *Roger Rabbit*, which is both a cartoon and a murder-mystery, will have components in all three of the entity sets *Movies*, *Cartoons*, and *Murder-Mysteries*. The three components are connected into one entity by the *isa* relationships. Together, these components give the *Roger Rabbit* entity all four attributes of *Movies* plus the attribute *weapon* of entity set *Murder-Mysteries* and the relationship *Vocies* of entity set *Cartoons*. □

### 4.1.12 Exercises for Section 4.1

**Exercise 4.1.1:** Design a database for a bank, including information about customers and their accounts. Information about a customer includes their name, address, phone, and Social Security number. Accounts have numbers, types (e.g., savings, checking) and balances. Also record the customer(s) who own an account. Draw the E/R diagram for this database. Be sure to include arrows where appropriate, to indicate the multiplicity of a relationship.

**Exercise 4.1.2:** Modify your solution to Exercise 4.1.1 as follows:

- Change your diagram so an account can have only one customer.
- Further change your diagram so a customer can have only one account.

- ! c) Change your original diagram of Exercise 4.1.1 so that a customer can have a set of addresses (which are street-city-state triples) and a set of phones. Remember that we do not allow attributes to have nonprimitive types, such as sets, in the E/R model.
- ! d) Further modify your diagram so that customers can have a set of addresses, and at each address there is a set of phones.

**Exercise 4.1.3:** Give an E/R diagram for a database recording information about teams, players, and their fans, including:

1. For each team, its name, its players, its team captain (one of its players), and the colors of its uniform.
2. For each player, his/her name.
3. For each fan, his/her name, favorite teams, favorite players, and favorite color.

Remember that a set of colors is not a suitable attribute type for teams. How can you get around this restriction?

**Exercise 4.1.4:** Suppose we wish to add to the schema of Exercise 4.1.3 a relationship *Led-by* among two players and a team. The intention is that this relationship set consists of triples (player1, player2, team) such that player 1 played on the team at a time when some other player 2 was the team captain.

- a) Draw the modification to the E/R diagram.
  - b) Replace your ternary relationship with a new entity set and binary relationships.
- ! c) Are your new binary relationships the same as any of the previously existing relationships? Note that we assume the two players are different, i.e., the team captain is not self-led.

**Exercise 4.1.5:** Modify Exercise 4.1.3 to record for each player the history of teams on which they have played, including the start date and ending date (if they were traded) for each such team.

**Exercise 4.1.6:** Design a genealogy database with one entity set: *People*. The information to record about persons includes their name (an attribute), their mother, father, and children.

**Exercise 4.1.7:** Modify your “people” database design of Exercise 4.1.6 to include the following special types of people:

1. Females.

2. Males.
3. People who are parents.

You may wish to distinguish certain other kinds of people as well, so relationships connect appropriate subclasses of people.

**Exercise 4.1.8:** An alternative way to represent the information of Exercise 4.1.6 is to have a ternary relationship *Family* with the intent that in the relationship set for *Family*, triple (person, mother, father) is a person, their mother, and their father; all three are in the *People* entity set, of course.

- a) Draw this diagram, placing arrows on edges where appropriate.
- b) Replace the ternary relationship *Family* by an entity set and binary relationships. Again place arrows to indicate the multiplicity of relationships.

**Exercise 4.1.9:** Design a database suitable for a university registrar. This database should include information about students, departments, professors, courses, which students are enrolled in which courses, which professors are teaching which courses, student grades, TA's for a course (TA's are students), which courses a department offers, and any other information you deem appropriate. Note that this question is more free-form than the questions above, and you need to make some decisions about multiplicities of relationships, appropriate types, and even what information needs to be represented.

! **Exercise 4.1.10:** Informally, we can say that two E/R diagrams “have the same information” if, given a real-world situation, the instances of these two diagrams that reflect this situation can be computed from one another. Consider the E/R diagram of Fig. 4.6. This four-way relationship can be decomposed into a three-way relationship and a binary relationship by taking advantage of the fact that for each movie, there is a unique studio that produces that movie. Give an E/R diagram without a four-way relationship that has the same information as Fig. 4.6.

## 4.2 Design Principles

We have yet to learn many of the details of the E/R model, but we have enough to begin study of the crucial issue of what constitutes a good design and what should be avoided. In this section, we offer some useful design principles.

### 4.2.1 Faithfulness

First and foremost, the design should be faithful to the specifications of the application. That is, entity sets and their attributes should reflect reality. You can't attach an attribute *number-of-cylinders* to *Stars*, although that attribute

would make sense for an entity set *Automobiles*. Whatever relationships are asserted should make sense given what we know about the part of the real world being modeled.

**Example 4.12:** If we define a relationship *Stars-in* between *Stars* and *Movies*, it should be a many-many relationship. The reason is that an observation of the real world tells us that stars can appear in more than one movie, and movies can have more than one star. It is incorrect to declare the relationship *Stars-in* to be many-one in either direction or to be one-one. □

**Example 4.13:** On the other hand, sometimes it is less obvious what the real world requires us to do in our E/R design. Consider, for instance, entity sets *Courses* and *Instructors*, with a relationship *Teaches* between them. Is *Teaches* many-one from *Courses* to *Instructors*? The answer lies in the policy and intentions of the organization creating the database. It is possible that the school has a policy that there can be only one instructor for any course. Even if several instructors may “team-teach” a course, the school may require that exactly one of them be listed in the database as the instructor responsible for the course. In either of these cases, we would make *Teaches* a many-one relationship from *Courses* to *Instructors*.

Alternatively, the school may use teams of instructors regularly and wish its database to allow several instructors to be associated with a course. Or, the intent of the *Teaches* relationship may not be to reflect the current teacher of a course, but rather those who have ever taught the course, or those who are capable of teaching the course; we cannot tell simply from the name of the relationship. In either of these cases, it would be proper to make *Teaches* be many-many. □

## 4.2.2 Avoiding Redundancy

We should be careful to say everything once only. The problems we discussed in Section 3.3 regarding redundancy and anomalies are typical of problems that can arise in E/R designs. However, in the E/R model, there are several new mechanisms whereby redundancy and other anomalies can arise.

For instance, we have used a relationship *Owns* between movies and studios. We might also choose to have an attribute *studioName* of entity set *Movies*. While there is nothing illegal about doing so, it is dangerous for several reasons.

1. Doing so leads to repetition of a fact, with the result that extra space is required to represent the data, once we convert the E/R design to a relational (or other type of) concrete implementation.
2. There is an update-anomaly potential, since we might change the relationship but not the attribute, or vice-versa.

We shall say more about avoiding anomalies in Sections 4.2.4 and 4.2.5.

### 4.2.3 Simplicity Counts

Avoid introducing more elements into your design than is absolutely necessary.

**Example 4.14:** Suppose that instead of a relationship between *Movies* and *Studios* we postulated the existence of “movie-holdings,” the ownership of a single movie. We might then create another entity set *Holdings*. A one-one relationship *Represents* could be established between each movie and the unique holding that represents the movie. A many-one relationship from *Holdings* to *Studios* completes the picture shown in Fig. 4.11.

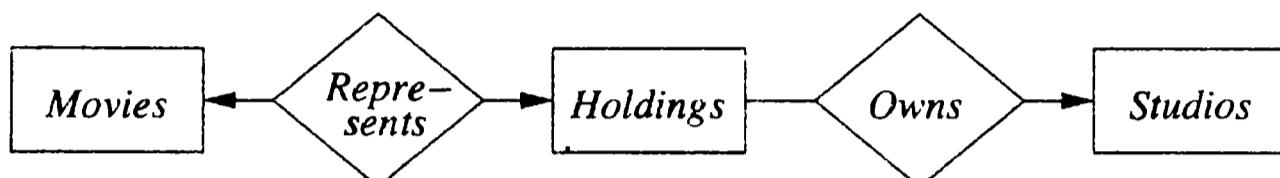


Figure 4.11: A poor design with an unnecessary entity set

Technically, the structure of Fig. 4.11 truly represents the real world, since it is possible to go from a movie to its unique owning studio via *Holdings*. However, *Holdings* serves no useful purpose, and we are better off without it. It makes programs that use the movie-studio relationship more complicated, wastes space, and encourages errors. □

### 4.2.4 Choosing the Right Relationships

Entity sets can be connected in various ways by relationships. However, adding to our design every possible relationship is not often a good idea. Doing so can lead to redundancy, update anomalies, and deletion anomalies, where the connected pairs or sets of entities for one relationship can be deduced from one or more other relationships. We shall illustrate the problem and what to do about it with two examples. In the first example, several relationships could represent the same information; in the second, one relationship could be deduced from several others.

**Example 4.15:** Let us review Fig. 4.7, where we connected movies, stars, and studios with a three-way relationship *Contracts*. We omitted from that figure the two binary relationships *Stars-in* and *Owns* from Fig. 4.2. Do we also need these relationships, between *Movies* and *Stars*, and between *Movies* and *Studios*, respectively? The answer is: “we don’t know; it depends on our assumptions regarding the three relationships in question.”

It might be possible to deduce the relationship *Stars-in* from *Contracts*. If a star can appear in a movie only if there is a contract involving that star, that movie, and the owning studio for the movie, then there truly is no need for relationship *Stars-in*. We could figure out all the star-movie pairs by looking at the star-movie-studio triples in the relationship set for *Contracts* and taking only the star and movie components, i.e., projecting *Contracts* onto *Stars-in*.

However, if a star can work on a movie without there being a contract — or what is more likely, without there being a contract that we know about in our database — then there could be star-movie pairs in *Stars-in* that are not part of star-movie-studio triples in *Contracts*. In that case, we need to retain the *Stars-in* relationship.

A similar observation applies to relationship *Owns*. If for every movie, there is at least one contract involving that movie, its owning studio, and some star for that movie, then we can dispense with *Owns*. However, if there is the possibility that a studio owns a movie, yet has no stars under contract for that movie, or no such contract is known to our database, then we must retain *Owns*.

In summary, we cannot tell you whether a given relationship will be redundant. You must find out from those who wish the database implemented what to expect. Only then can you make a rational decision about whether or not to include relationships such as *Stars-in* or *Owns*. □

**Example 4.16:** Now, consider Fig. 4.2 again. In this diagram, there is no relationship between stars and studios. Yet we can use the two relationships *Stars-in* and *Owns* to build a connection by the process of composing those two relationships. That is, a star is connected to some movies by *Stars-in*, and those movies are connected to studios by *Owns*. Thus, we could say that a star is connected to the studios that own movies in which the star has appeared.

Would it make sense to have a relationship *Works-for*, as suggested in Fig. 4.12, between *Stars* and *Studios* too? Again, we cannot tell without knowing more. First, what would the meaning of this relationship be? If it is to mean “the star appeared in at least one movie of this studio,” then probably there is no good reason to include it in the diagram. We could deduce this information from *Stars-in* and *Owns* instead.

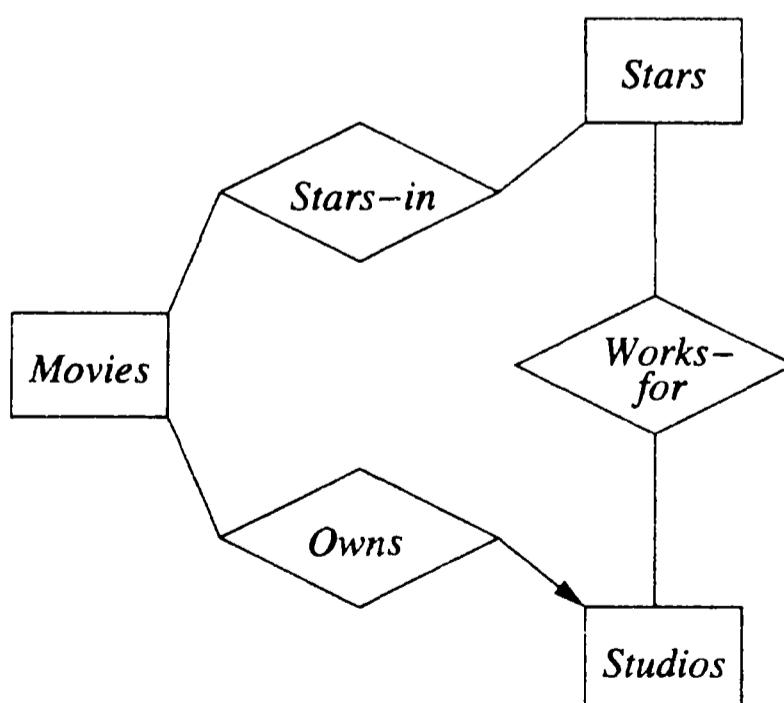


Figure 4.12: Adding a relationship between *Stars* and *Studios*

However, perhaps we have other information about stars working for studios that is not implied by the connection through a movie. In that case, a

relationship connecting stars directly to studios might be useful and would not be redundant. Alternatively, we might use a relationship between stars and studios to mean something entirely different. For example, it might represent the fact that the star is under contract to the studio, in a manner unrelated to any movie. As we suggested in Example 4.7, it is possible for a star to be under contract to one studio and yet work on a movie owned by another studio. In this case, the information found in the new *Works-for* relation would be independent of the *Stars-in* and *Owns* relationships, and would surely be nonredundant.  $\square$

#### 4.2.5 Picking the Right Kind of Element

Sometimes we have options regarding the type of design element used to represent a real-world concept. Many of these choices are between using attributes and using entity set/relationship combinations. In general, an attribute is simpler to implement than either an entity set or a relationship. However, making everything an attribute will usually get us into trouble.

**Example 4.17:** Let us consider a specific problem. In Fig. 4.2, were we wise to make studios an entity set? Should we instead have made the name and address of the studio be attributes of movies and eliminated the *Studio* entity set? One problem with doing so is that we repeat the address of the studio for each movie. We can also have an update anomaly if we change the address for one movie but not another with the same studio, and we can have a deletion anomaly if we delete the last movie owned by a given studio.

On the other hand, if we did not record addresses of studios, then there is no harm in making the studio name an attribute of movies. We have no anomalies in this case. Saying the name of a studio for each movie is not true redundancy, since we must represent the owner of each movie somehow, and saying the name of the studio is a reasonable way to do so.  $\square$

We can abstract what we have observed in Example 4.17 to give the conditions under which we prefer to use an attribute instead of an entity set. Suppose  $E$  is an entity set. Here are conditions that  $E$  must obey in order for us to replace  $E$  by an attribute or attributes of several other entity sets.

1. All relationships in which  $E$  is involved must have arrows entering  $E$ . That is,  $E$  must be the “one” in many-one relationships, or its generalization for the case of multiway relationships.
2. If  $E$  has more than one attribute, then no attribute depends on the other attributes, the way *address* depends on *name* for *Studios*. That is, the only key for  $E$  is all its attributes.
3. No relationship involves  $E$  more than once.

If these conditions are met, then we can replace entity set  $E$  as follows:

- a) If there is a many-one relationship  $R$  from some entity set  $F$  to  $E$ , then remove  $R$  and make the attributes of  $E$  be attributes of  $F$ , suitably renamed if they conflict with attribute names for  $F$ . In effect, each  $F$ -entity takes, as attributes, the name of the unique, related  $E$ -entity.<sup>2</sup> For instance, *Movies* entities could take their studio name as an attribute, should we dispense with studio addresses.
- b) If there is a multiway relationship  $R$  with an arrow to  $E$ , make the attributes of  $E$  be attributes of  $R$  and delete the arc from  $R$  to  $E$ . An example of this transformation is replacing Fig. 4.8, where there is an entity set *Salaries* with a number as its lone attribute, by its original diagram in Fig. 4.7.

**Example 4.18:** Let us consider a point where there is a tradeoff between using a multiway relationship and using a connecting entity set with several binary relationships. We saw a four-way relationship *Contracts* among a star, a movie, and two studios in Fig. 4.6. In Fig. 4.9, we mechanically converted it to an entity set *Contracts*. Does it matter which we choose?

As the problem was stated, either is appropriate. However, should we change the problem just slightly, then we are almost forced to choose a connecting entity set. Let us suppose that contracts involve one star, one movie, but any set of studios. This situation is more complex than the one in Fig. 4.6, where we had two studios playing two roles. In this case, we can have any number of studios involved, perhaps one to do production, one for special effects, one for distribution, and so on. Thus, we cannot assign roles for studios.

It appears that a relationship set for the relationship *Contracts* must contain triples of the form (star, movie, set-of-studios), and the relationship *Contracts* itself involves not only the usual *Stars* and *Movies* entity sets, but a new entity set whose entities are *sets of* studios. While this approach is possible, it seems unnatural to think of sets of studios as basic entities, and we do not recommend it.

A better approach is to think of contracts as an entity set. As in Fig. 4.9, a contract entity connects a star, a movie and a set of studios, but now there must be no limit on the number of studios. Thus, the relationship between contracts and studios is many-many, rather than many-one as it would be if contracts were a true “connecting” entity set. Figure 4.13 sketches the E/R diagram. Note that a contract is related to a single star and to a single movie, but to any number of studios. □

#### 4.2.6 Exercises for Section 4.2

**Exercise 4.2.1:** In Fig. 4.14 is an E/R diagram for a bank database involving customers and accounts. Since customers may have several accounts, and

---

<sup>2</sup>In a situation where an  $F$ -entity is not related to any  $E$ -entity, the new attributes of  $F$  would be given special “null” values to indicate the absence of a related  $E$ -entity. A similar arrangement would be used for the new attributes of  $R$  in case (b).

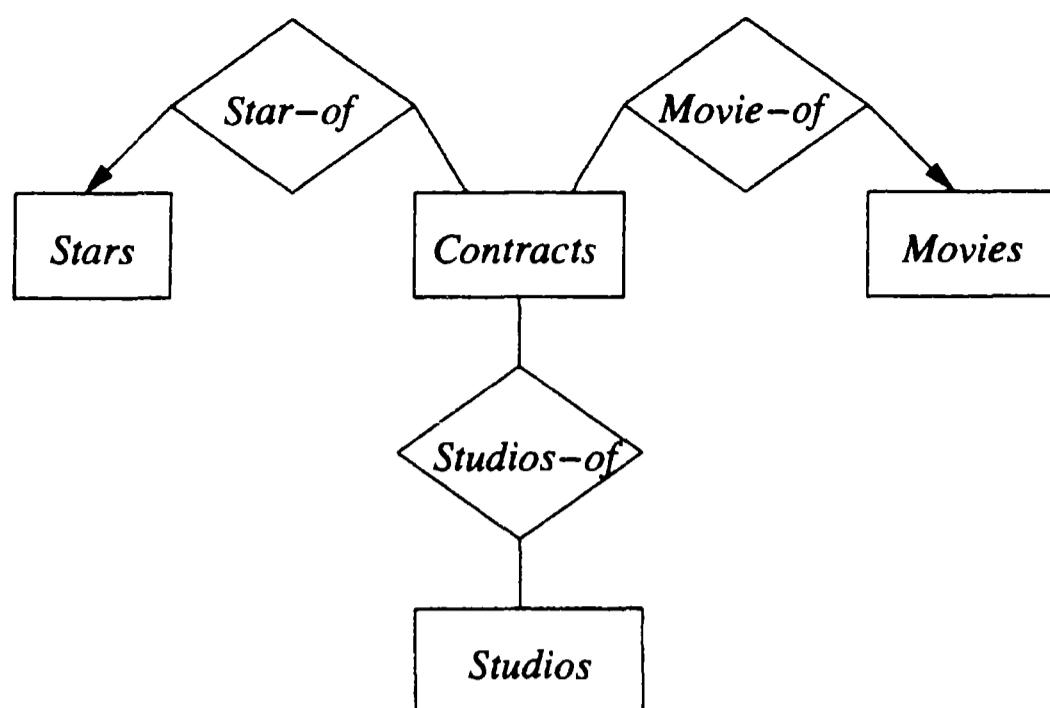


Figure 4.13: Contracts connecting a star, a movie, and a set of studios

accounts may be held jointly by several customers, we associate with each customer an “account set,” and accounts are members of one or more account sets. Assuming the meaning of the various relationships and attributes are as expected given their names, criticize the design. What design rules are violated? Why? What modifications would you suggest?

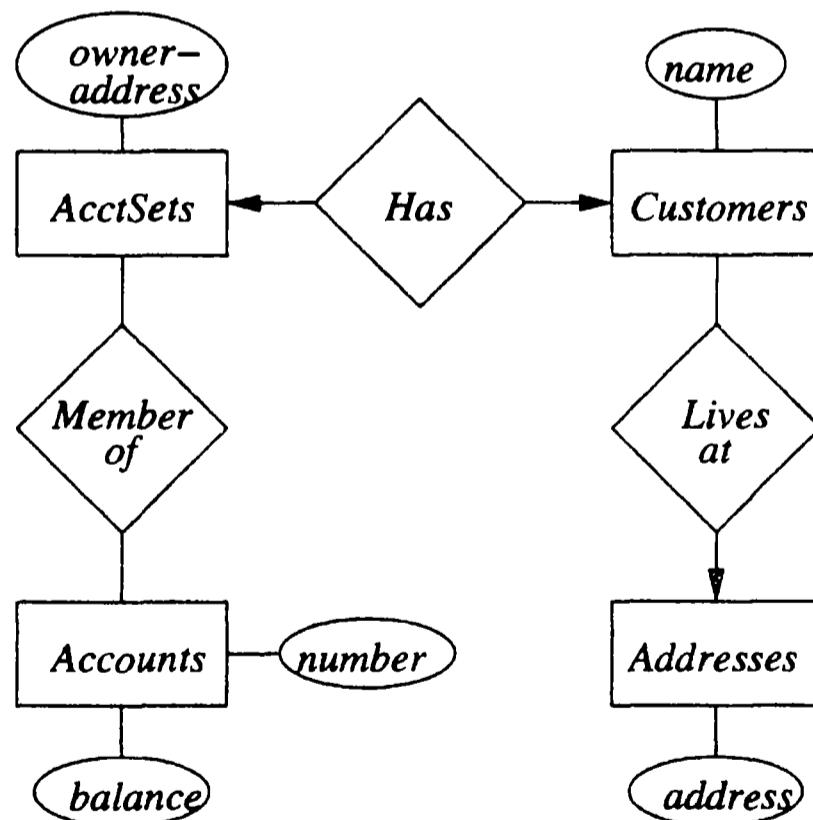


Figure 4.14: A poor design for a bank database

**Exercise 4.2.2:** Under what circumstances (regarding the unseen attributes of *Studios* and *Presidents*) would you recommend combining the two entity sets and relationship in Fig. 4.3 into a single entity set and attributes?

**Exercise 4.2.3:** Suppose we delete the attribute *address* from *Studios* in Fig. 4.7. Show how we could then replace an entity set by an attribute. Where

would that attribute appear?

**Exercise 4.2.4:** Give choices of attributes for the following entity sets in Fig. 4.13 that will allow the entity set to be replaced by an attribute:

- a) *Stars*.
- b) *Movies*.
- c) *Studios*.

!! **Exercise 4.2.5:** In this and following exercises we shall consider two design options in the E/R model for describing births. At a birth, there is one baby (twins would be represented by two births), one mother, any number of nurses, and any number of doctors. Suppose, therefore, that we have entity sets *Babies*, *Mothers*, *Nurses*, and *Doctors*. Suppose we also use a relationship *Births*, which connects these four entity sets, as suggested in Fig. 4.15. Note that a tuple of the relationship set for *Births* has the form (baby, mother, nurse, doctor). If there is more than one nurse and/or doctor attending a birth, then there will be several tuples with the same baby and mother, one for each combination of nurse and doctor.

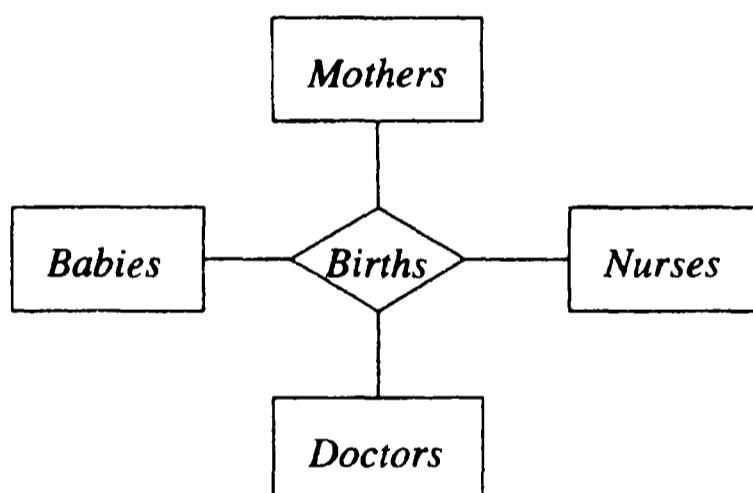


Figure 4.15: Representing births by a multiway relationship

There are certain assumptions that we might wish to incorporate into our design. For each, tell how to add arrows or other elements to the E/R diagram in order to express the assumption.

- a) For every baby, there is a unique mother.
- b) For every combination of a baby, nurse, and doctor, there is a unique mother.
- c) For every combination of a baby and a mother there is a unique doctor.

! **Exercise 4.2.6:** Another approach to the problem of Exercise 4.2.5 is to connect the four entity sets *Babies*, *Mothers*, *Nurses*, and *Doctors* by an entity set

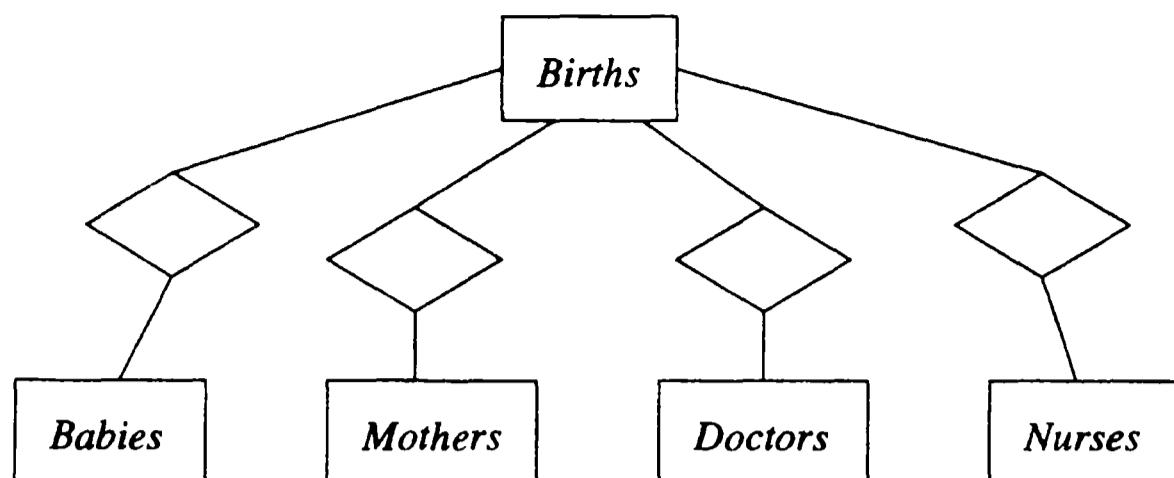


Figure 4.16: Representing births by an entity set

*Births*, with four relationships, one between *Births* and each of the other entity sets, as suggested in Fig. 4.16. Use arrows (indicating that certain of these relationships are many-one) to represent the following conditions:

- Every baby is the result of a unique birth, and every birth is of a unique baby.
- In addition to (a), every baby has a unique mother.
- In addition to (a) and (b), for every birth there is a unique doctor.

In each case, what design flaws do you see?

!! **Exercise 4.2.7:** Suppose we change our viewpoint to allow a birth to involve more than one baby born to one mother. How would you represent the fact that every baby still has a unique mother using the approaches of Exercises 4.2.5 and 4.2.6?

## 4.3 Constraints in the E/R Model

The E/R model has several ways to express the common kinds of constraints on the data that will populate the database being designed. Like the relational model, there is a way to express the idea that an attribute or attributes are a key for an entity set. We have already seen how an arrow connecting a relationship to an entity set serves as a “functional dependency.” There is also a way to express a referential-integrity constraint, where an entity in one set is required to have an entity in another set to which it is related.

### 4.3.1 Keys in the E/R Model

A *key* for an entity set  $E$  is a set  $K$  of one or more attributes such that, given any two distinct entities  $e_1$  and  $e_2$  in  $E$ ,  $e_1$  and  $e_2$  cannot have identical values for each of the attributes in the key  $K$ . If  $K$  consists of more than one attribute, then it is possible for  $e_1$  and  $e_2$  to agree in some of these attributes, but never in all attributes. Some important points to remember are:

- Every entity set must have a key, although in some cases — isa-hierarchies and “weak” entity sets (see Section 4.4), the key actually belongs to another entity set.
- There can be more than one possible key for an entity set. However, it is customary to pick one key as the “primary key,” and to act as if that were the only key.
- When an entity set is involved in an isa-hierarchy, we require that the root entity set have all the attributes needed for a key, and that the key for each entity is found from its component in the root entity set, regardless of how many entity sets in the hierarchy have components for the entity.

In our running movies example, we have used *title* and *year* as the key for *Movies*, counting on the observation that it is unlikely that two movies with the same title would be released in one year. We also decided that it was safe to use *name* as a key for *MovieStar*, believing that no real star would ever want to use the name of another star.

### 4.3.2 Representing Keys in the E/R Model

In our E/R-diagram notation, we underline the attributes belonging to a key for an entity set. For example, Fig. 4.17 reproduces our E/R diagram for movies, stars, and studios from Fig. 4.2, but with key attributes underlined. Attribute *name* is the key for *Stars*. Likewise, *Studios* has a key consisting of only its own attribute *name*.

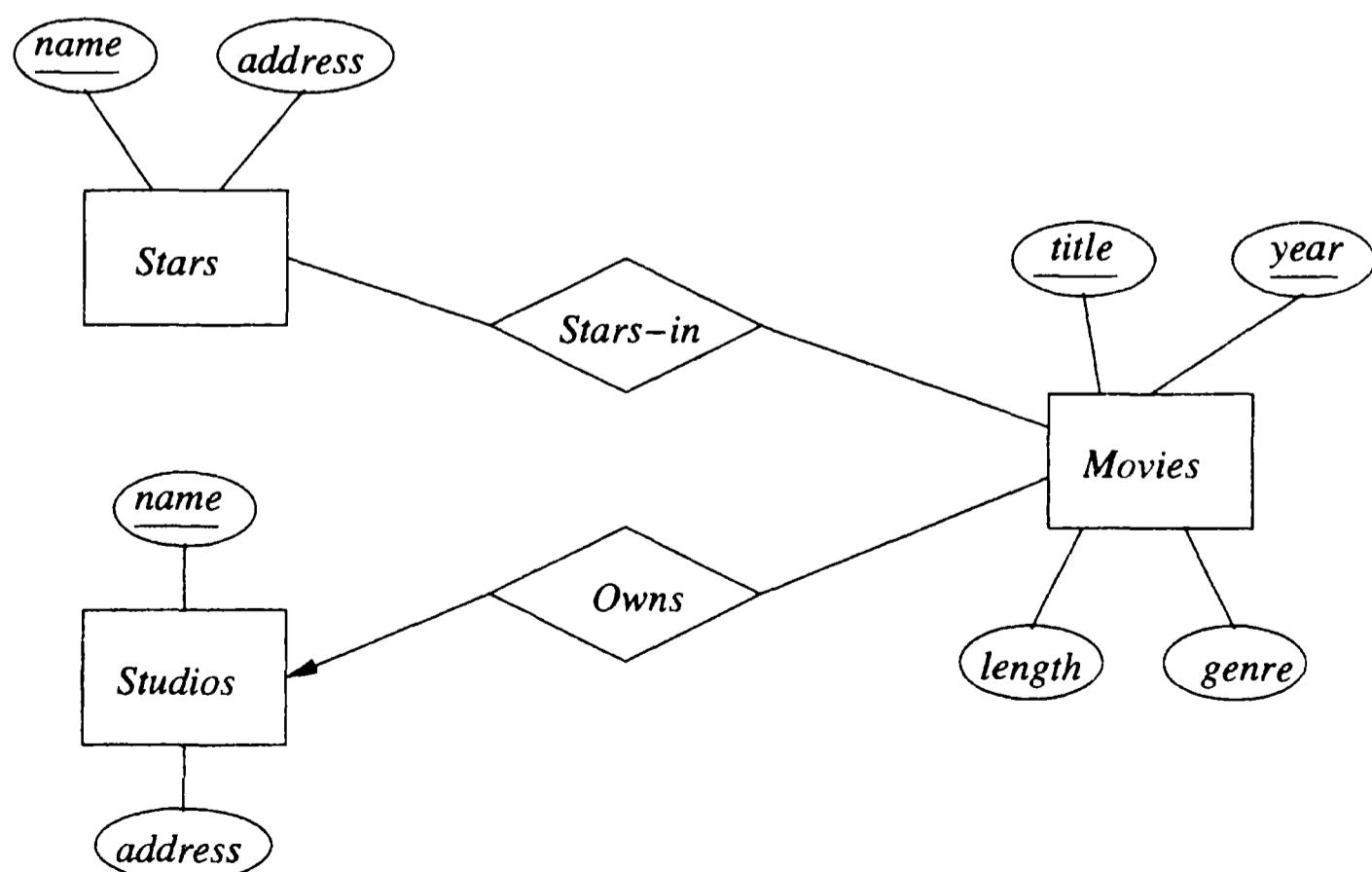


Figure 4.17: E/R diagram; keys are indicated by underlines

The attributes *title* and *year* together form the key for *Movies*. Note that when several attributes are underlined, as in Fig. 4.17, then they are each members of the key. There is no notation for representing the situation where there are several keys for an entity set; we underline only the primary key. You should also be aware that in some unusual situations, the attributes forming the key for an entity set do not all belong to the entity set itself. We shall defer this matter, called “weak entity sets,” until Section 4.4.

### 4.3.3 Referential Integrity

Recall our discussion of referential-integrity constraints in Section 2.5.2. These constraints say that a value appearing in one context must also appear in another. For example, let us consider the many-one relationship *Owns* from *Movies* to *Studios* in Fig. 4.2. The many-one requirement simply says that no movie can be owned by more than one studio. It does *not* say that a movie must surely be owned by a studio, or that the owning studio must be present in the *Studios* entity set, as stored in our database. An appropriate referential integrity constraint on relationship *Owns* is that for each movie, the owning studio (the entity “referenced” by the relationship for this movie) must exist in our database.

The arrow notation in E/R diagrams is able to indicate whether a relationship is expected to support referential integrity in one or more directions. Suppose *R* is a relationship from entity set *E* to entity set *F*. A rounded arrow-head pointing to *F* indicates not only that the relationship is many-one from *E* to *F*, but that the entity of set *F* related to a given entity of set *E* is required to exist. The same idea applies when *R* is a relationship among more than two entity sets.

**Example 4.19:** Figure 4.18 shows some appropriate referential integrity constraints among the entity sets *Movies*, *Studios*, and *Presidents*. These entity sets and relationships were first introduced in Figs. 4.2 and 4.3. We see a rounded arrow entering *Studios* from relationship *Owns*. That arrow expresses the referential integrity constraint that every movie must be owned by one studio, and this studio is present in the *Studios* entity set.



Figure 4.18: E/R diagram showing referential integrity constraints

Similarly, we see a rounded arrow entering *Studios* from *Runs*. That arrow expresses the referential integrity constraint that every president runs a studio that exists in the *Studios* entity set.

Note that the arrow to *Presidents* from *Runs* remains a pointed arrow. That choice reflects a reasonable assumption about the relationship between studios

and their presidents. If a studio ceases to exist, its president can no longer be called a president, so we would expect the president of the studio to be deleted from the entity set *Presidents*. Hence there is a rounded arrow to *Studios*. On the other hand, if a president were fired or resigned, the studio would continue to exist. Thus, we place an ordinary, pointed arrow to *Presidents*, indicating that each studio has at most one president, but might have no president at some time.  $\square$

#### 4.3.4 Degree Constraints

In the E/R model, we can attach a bounding number to the edges that connect a relationship to an entity set, indicating limits on the number of entities that can be connected to any one entity of the related entity set. For example, we could choose to place a constraint on the degree of a relationship, such as that a movie entity cannot be connected by relationship *Stars-in* to more than 10 star entities.



Figure 4.19: Representing a constraint on the number of stars per movie

Figure 4.19 shows how we can represent this constraint. As another example, we can think of the arrow as a synonym for the constraint " $\leq 1$ ," and we can think of the rounded arrow of Fig. 4.18 as standing for the constraint " $= 1$ ."

#### 4.3.5 Exercises for Section 4.3

**Exercise 4.3.1:** For your E/R diagrams of:

- a) Exercise 4.1.1.
- b) Exercise 4.1.3.
- c) Exercise 4.1.6.

(i) Select and specify keys, and (ii) Indicate appropriate referential integrity constraints.

**Exercise 4.3.2:** We may think of relationships in the E/R model as having keys, just as entity sets do. Let  $R$  be a relationship among the entity sets  $E_1, E_2, \dots, E_n$ . Then a *key* for  $R$  is a set  $K$  of attributes chosen from the attributes of  $E_1, E_2, \dots, E_n$  such that if  $(e_1, e_2, \dots, e_n)$  and  $(f_1, f_2, \dots, f_n)$  are two different tuples in the relationship set for  $R$ , then it is not possible that these tuples agree in all the attributes of  $K$ . Now, suppose  $n = 2$ ; that is,  $R$  is a binary relationship. Also, for each  $i$ , let  $K_i$  be a set of attributes that is a key for entity set  $E_i$ . In terms of  $E_1$  and  $E_2$ , give a smallest possible key for  $R$  under the assumption that:

- a)  $R$  is many-many.
- b)  $R$  is many-one from  $E_1$  to  $E_2$ .
- c)  $R$  is many-one from  $E_2$  to  $E_1$ .
- d)  $R$  is one-one.

**!! Exercise 4.3.3:** Consider again the problem of Exercise 4.3.2, but with  $n$  allowed to be any number, not just 2. Using only the information about which arcs from  $R$  to the  $E_i$ 's have arrows, show how to find a smallest possible key  $K$  for  $R$  in terms of the  $K_i$ 's.

## 4.4 Weak Entity Sets

It is possible for an entity set's key to be composed of attributes, some or all of which belong to another entity set. Such an entity set is called a *weak entity set*.

### 4.4.1 Causes of Weak Entity Sets

There are two principal reasons we need weak entity sets. First, sometimes entity sets fall into a hierarchy based on classifications unrelated to the “isa hierarchy” of Section 4.1.11. If entities of set  $E$  are subunits of entities in set  $F$ , then it is possible that the names of  $E$ -entities are not unique until we take into account the name of the  $F$ -entity to which the  $E$  entity is subordinate. Several examples will illustrate the problem.

**Example 4.20:** A movie studio might have several film crews. The crews might be designated by a given studio as crew 1, crew 2, and so on. However, other studios might use the same designations for crews, so the attribute *number* is not a key for crews. Rather, to name a crew uniquely, we need to give both the name of the studio to which it belongs and the number of the crew. The situation is suggested by Fig. 4.20. The double-rectangle indicates a weak entity set, and the double-diamond indicates a many-one relationship that helps provide the key for the weak entity set. The notation will be explained further in Section 4.4.3. The key for weak entity set *Crews* is its own *number* attribute and the *name* attribute of the unique studio to which the crew is related by the many-one *Unit-of* relationship.  $\square$

**Example 4.21:** A species is designated by its genus and species names. For example, humans are of the species *Homo sapiens*; *Homo* is the genus name and *sapiens* the species name. In general, a genus consists of several species, each of which has a name beginning with the genus name and continuing with the species name. Unfortunately, species names, by themselves, are not unique.

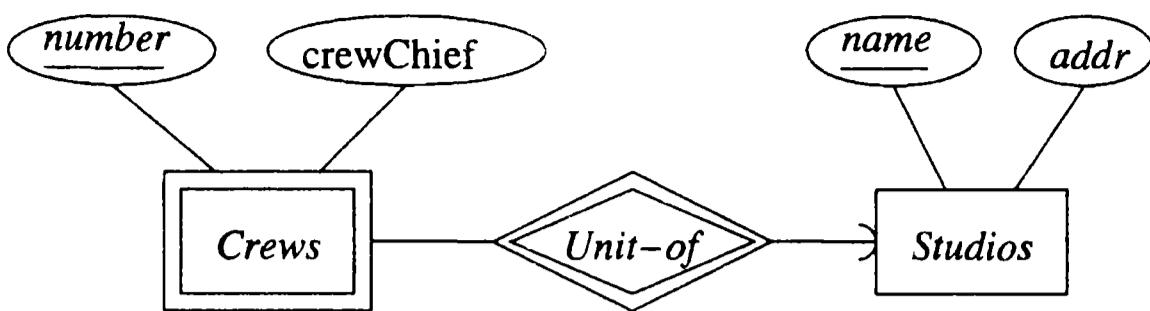


Figure 4.20: A weak entity set for crews, and its connections

Two or more genera may have species with the same species name. Thus, to designate a species uniquely we need both the species name and the name of the genus to which the species is related by the *Belongs-to* relationship, as suggested in Fig. 4.21. *Species* is a weak entity set whose key comes partially from its genus. □

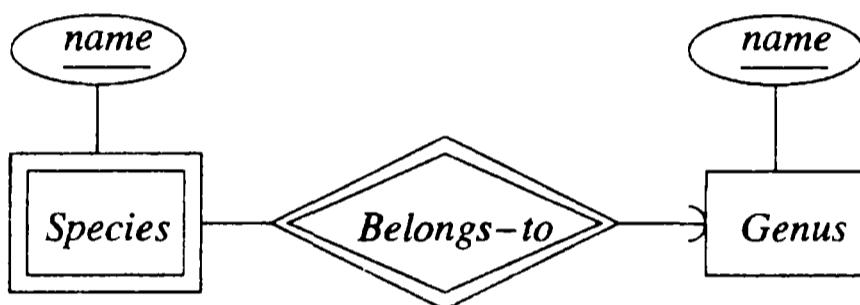


Figure 4.21: Another weak entity set, for species

The second common source of weak entity sets is the connecting entity sets that we introduced in Section 4.1.10 as a way to eliminate a multiway relationship.<sup>3</sup> These entity sets often have no attributes of their own. Their key is formed from the attributes that are the key attributes for the entity sets they connect.

**Example 4.22:** In Fig. 4.22 we see a connecting entity set *Contracts* that replaces the ternary relationship *Contracts* of Example 4.5. *Contracts* has an attribute *salary*, but this attribute does not contribute to the key. Rather, the key for a contract consists of the name of the studio and the star involved, plus the title and year of the movie involved. □

#### 4.4.2 Requirements for Weak Entity Sets

We cannot obtain key attributes for a weak entity set indiscriminately. Rather, if  $E$  is a weak entity set then its key consists of:

1. Zero or more of its own attributes, and

---

<sup>3</sup>Remember that there is no particular requirement in the E/R model that multiway relationships be eliminated, although this requirement exists in some other database design models.

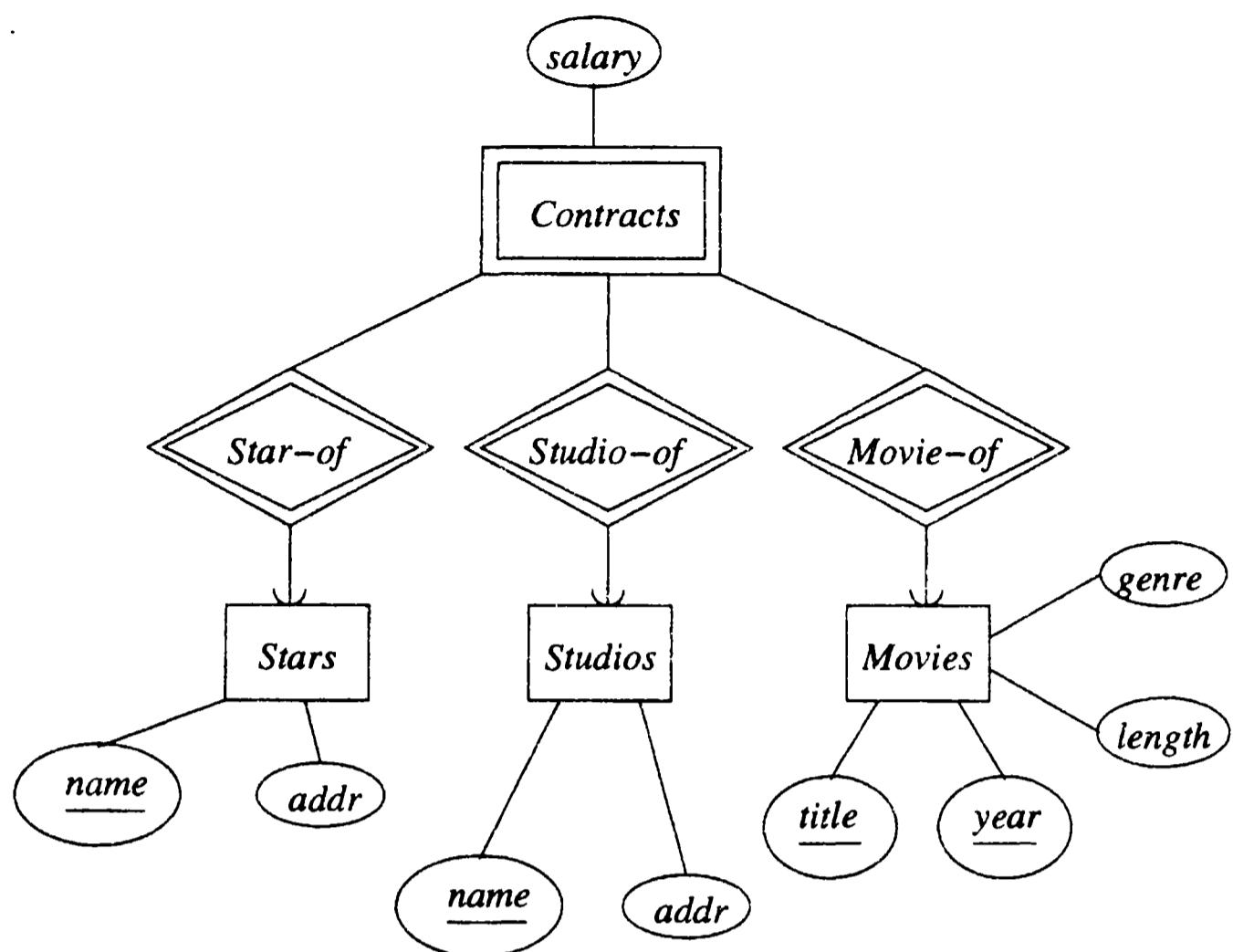


Figure 4.22: Connecting entity sets are weak

2. Key attributes from entity sets that are reached by certain many-one relationships from  $E$  to other entity sets. These many-one relationships are called *supporting relationships* for  $E$ , and the entity sets reached from  $E$  are *supporting entity sets*.

In order for  $R$ , a many-one relationship from  $E$  to some entity set  $F$ , to be a supporting relationship for  $E$ , the following conditions must be obeyed:

- a)  $R$  must be a binary, many-one relationship<sup>4</sup> from  $E$  to  $F$ .
- b)  $R$  must have referential integrity from  $E$  to  $F$ . That is, for every  $E$ -entity, there must be exactly one existing  $F$ -entity related to it by  $R$ . Put another way, a rounded arrow from  $R$  to  $F$  must be justified.
- c) The attributes that  $F$  supplies for the key of  $E$  must be key attributes of  $F$ .
- d) However, if  $F$  is itself weak, then some or all of the key attributes of  $F$  supplied to  $E$  will be key attributes of one or more entity sets  $G$  to which  $F$  is connected by a supporting relationship. Recursively, if  $G$  is weak, some key attributes of  $G$  will be supplied from elsewhere, and so on.

<sup>4</sup>Remember that a one-one relationship is a special case of a many-one relationship. When we say a relationship must be many-one, we always include one-one relationships as well.

- e) If there are several different supporting relationships from  $E$  to the same entity set  $F$ , then each relationship is used to supply a copy of the key attributes of  $F$  to help form the key of  $E$ . Note that an entity  $e$  from  $E$  may be related to different entities in  $F$  through different supporting relationships from  $E$ . Thus, the keys of several different entities from  $F$  may appear in the key values identifying a particular entity  $e$  from  $E$ .

The intuitive reason why these conditions are needed is as follows. Consider an entity in a weak entity set, say a crew in Example 4.20. Each crew is unique, abstractly. In principle we can tell one crew from another, even if they have the same number but belong to different studios. It is only the data about crews that makes it hard to distinguish crews, because the number alone is not sufficient. The only way we can associate additional information with a crew is if there is some deterministic process leading to additional values that make the designation of a crew unique. But the only unique values associated with an abstract crew entity are:

1. Values of attributes of the *Crews* entity set, and
2. Values obtained by following a relationship from a crew entity to a unique entity of some other entity set, where that other entity has a unique associated value of some kind. That is, the relationship followed must be many-one to the other entity set  $F$ , and the associated value must be part of a key for  $F$ .

#### 4.4.3 Weak Entity Set Notation

We shall adopt the following conventions to indicate that an entity set is weak and to declare its key attributes.

1. If an entity set is weak, it will be shown as a rectangle with a double border. Examples of this convention are *Crews* in Fig. 4.20 and *Contracts* in Fig. 4.22.
2. Its supporting many-one relationships will be shown as diamonds with a double border. Examples of this convention are *Unit-of* in Fig. 4.20 and all three relationships in Fig. 4.22.
3. If an entity set supplies any attributes for its own key, then those attributes will be underlined. An example is in Fig. 4.20, where the number of a crew participates in its own key, although it is not the complete key for *Crews*.

We can summarize these conventions with the following rule:

- Whenever we use an entity set  $E$  with a double border, it is weak. The key for  $E$  is whatever attributes of  $E$  are underlined plus the key attributes of those entity sets to which  $E$  is connected by many-one relationships with a double border.

We should remember that the double-diamond is used only for supporting relationships. It is possible for there to be many-one relationships from a weak entity set that are not supporting relationships, and therefore do not get a double diamond.

**Example 4.23:** In Fig. 4.22, the relationship *Studio-of* need not be a supporting relationship for *Contracts*. The reason is that each movie has a unique owning studio, determined by the (not shown) many-one relationship from *Movies* to *Studios*. Thus, if we are told the name of a star and a movie, there is at most one contract with any studio for the work of that star in that movie. In terms of our notation, it would be appropriate to use an ordinary single diamond, rather than the double diamond, for *Studio-of* in Fig. 4.22. □

#### 4.4.4 Exercises for Section 4.4

**Exercise 4.4.1:** One way to represent students and the grades they get in courses is to use entity sets corresponding to students, to courses, and to “enrollments.” Enrollment entities form a “connecting” entity set between students and courses and can be used to represent not only the fact that a student is taking a certain course, but the grade of the student in the course. Draw an E/R diagram for this situation, indicating weak entity sets and the keys for the entity sets. Is the grade part of the key for enrollments?

**Exercise 4.4.2:** Modify your solution to Exercise 4.4.1 so that we can record grades of the student for each of several assignments within a course. Again, indicate weak entity sets and keys.

**Exercise 4.4.3:** For your E/R diagrams of Exercise 4.2.6(a)–(c), indicate weak entity sets, supporting relationships, and keys.

**Exercise 4.4.4:** Draw E/R diagrams for the following situations involving weak entity sets. In each case indicate keys for entity sets.

- a) Entity sets *Courses* and *Departments*. A course is given by a unique department, but its only attribute is its number. Different departments can offer courses with the same number. Each department has a unique name.
- b) Entity sets *Leagues*, *Teams*, and *Players*. League names are unique. No league has two teams with the same name. No team has two players with the same number. However, there can be players with the same number on different teams, and there can be teams with the same name in different leagues.

## 4.5 From E/R Diagrams to Relational Designs

To a first approximation, converting an E/R design to a relational database schema is straightforward:

- Turn each entity set into a relation with the same set of attributes, and
- Replace a relationship by a relation whose attributes are the keys for the connected entity sets.

While these two rules cover much of the ground, there are also several special situations that we need to deal with, including:

1. Weak entity sets cannot be translated straightforwardly to relations.
2. “Isa” relationships and subclasses require careful treatment.
3. Sometimes, we do well to combine two relations, especially the relation for an entity set  $E$  and the relation that comes from a many-one relationship from  $E$  to some other entity set.

### 4.5.1 From Entity Sets to Relations

Let us first consider entity sets that are not weak. We shall take up the modifications needed to accommodate weak entity sets in Section 4.5.4. For each non-weak entity set, we shall create a relation of the same name and with the same set of attributes. This relation will not have any indication of the relationships in which the entity set participates; we’ll handle relationships with separate relations, as discussed in Section 4.5.2.

**Example 4.24:** Consider the three entity sets *Movies*, *Stars* and *Studios* from Fig. 4.17, which we reproduce here as Fig. 4.23. The attributes for the *Movies* entity set are *title*, *year*, *length*, and *genre*. As a result, this relation *Movies* looks just like the relation *Movies* of Fig. 2.1 with which we began Section 2.2.

Next, consider the entity set *Stars* from Fig. 4.23. There are two attributes, *name* and *address*. Thus, we would expect the corresponding *Stars* relation to have schema *Stars(name, address)* and for

<i>name</i>	<i>address</i>
Carrie Fisher	123 Maple St., Hollywood
Mark Hamill	456 Oak Rd., Brentwood
Harrison Ford	789 Palm Dr., Beverly Hills

to be a typical instance.  $\square$

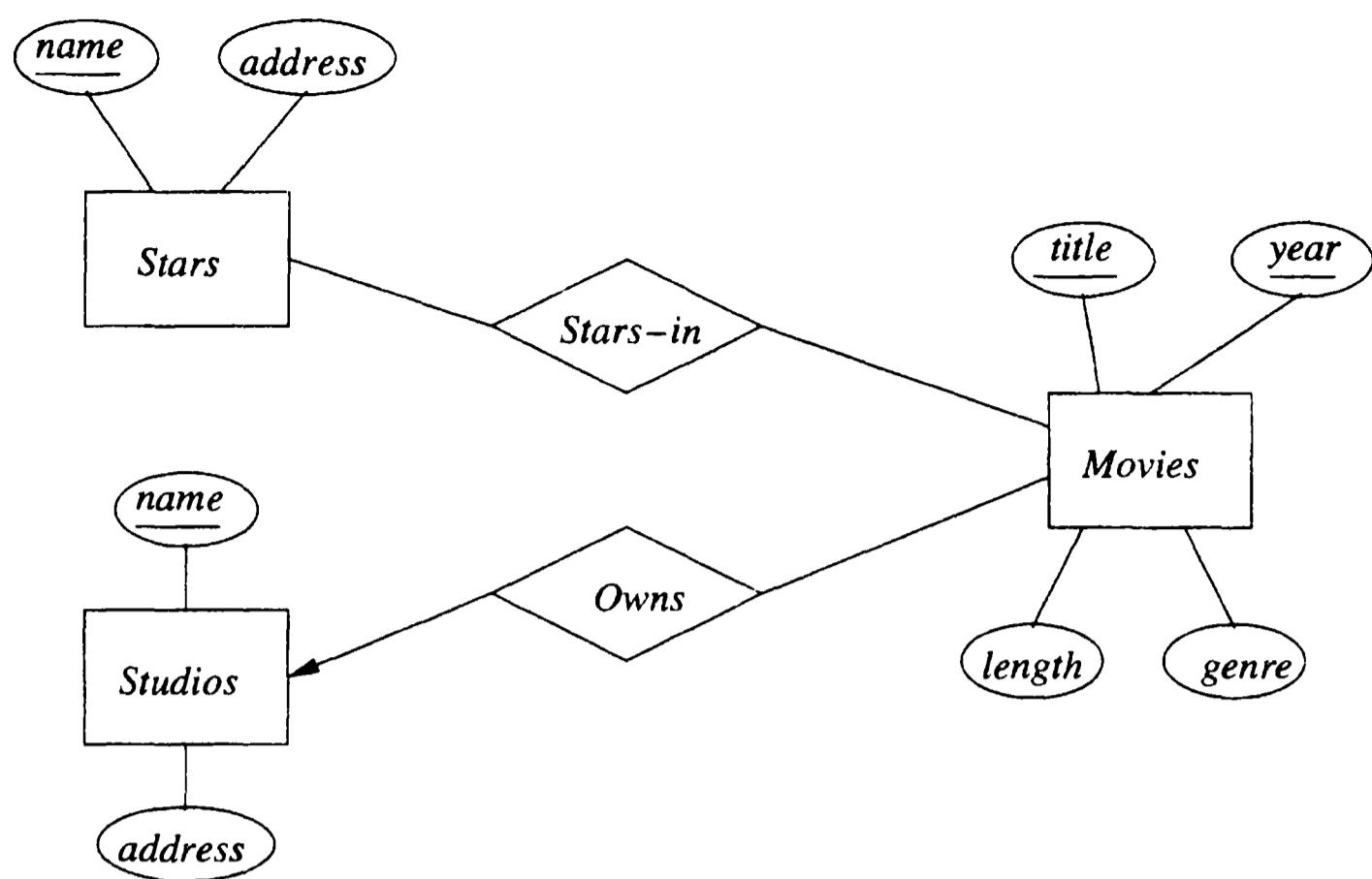


Figure 4.23: E/R diagram for the movie database

#### 4.5.2 From E/R Relationships to Relations

Relationships in the E/R model are also represented by relations. The relation for a given relationship  $R$  has the following attributes:

1. For each entity set involved in relationship  $R$ , we take its key attribute or attributes as part of the schema of the relation for  $R$ .
2. If the relationship has attributes, then these are also attributes of relation  $R$ .

If one entity set is involved several times in a relationship, in different roles, then its key attributes each appear as many times as there are roles. We must rename the attributes to avoid name duplication. More generally, should the same attribute name appear twice or more among the attributes of  $R$  itself and the keys of the entity sets involved in relationship  $R$ , then we need to rename to avoid duplication.

**Example 4.25 :** Consider the relationship *Owns* of Fig. 4.23. This relationship connects entity sets *Movies* and *Studios*. Thus, for the schema of relation *Owns* we use the key for *Movies*, which is *title* and *year*, and the key of *Studios*, which is *name*. That is, the schema for relation *Owns* is:

*Owns(title, year, studioName)*

A sample instance of this relation is:

<i>title</i>	<i>year</i>	<i>studioName</i>
Star Wars	1977	Fox
Gone With the Wind	1939	MGM
Wayne's World	1992	Paramount

We have chosen the attribute *studioName* for clarity; it corresponds to the attribute *name* of *Studios*. □

<i>title</i>	<i>year</i>	<i>starName</i>
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Gone With the Wind	1939	Vivien Leigh
Wayne's World	1992	Dana Carvey
Wayne's World	1992	Mike Meyers

Figure 4.24: A relation for relationship *Stars-In*

**Example 4.26:** Similarly, the relationship *Stars-In* of Fig. 4.23 can be transformed into a relation with the attributes *title* and *year* (the key for *Movies*) and attribute *starName*, which is the key for entity set *Stars*. Figure 4.24 shows a sample relation *Stars-In*. □

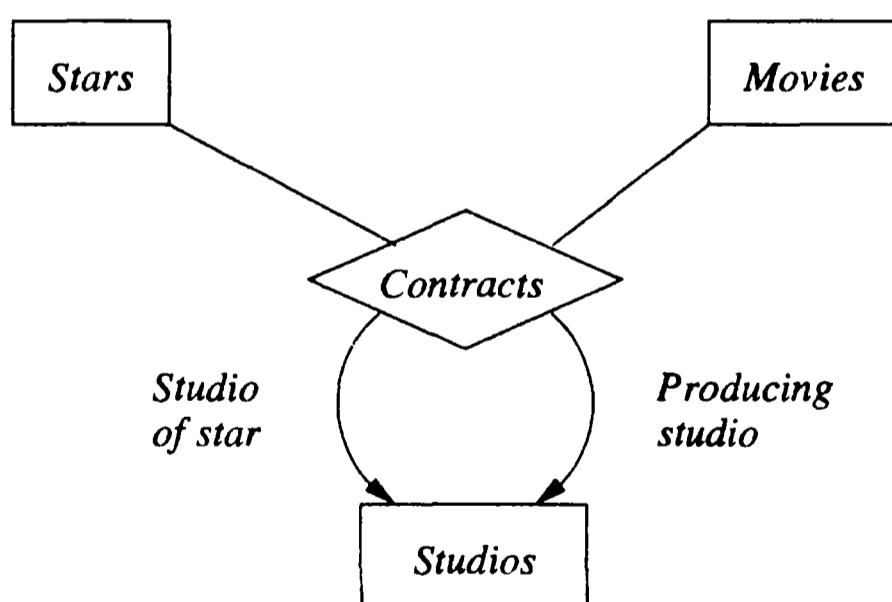


Figure 4.25: The relationship *Contracts*

**Example 4.27:** Multiway relationships are also easy to convert to relations. Consider the four-way relationship *Contracts* of Fig. 4.6, reproduced here as Fig. 4.25, involving a star, a movie, and two studios — the first holding the star's contract and the second contracting for that star's services in that movie. We represent this relationship by a relation *Contracts* whose schema consists of the attributes from the keys of the following four entity sets:

1. The key **starName** for the star.
2. The key consisting of attributes **title** and **year** for the movie.
3. The key **studioOfStar** indicating the name of the first studio; recall we assume the studio name is a key for the entity set **Studios**.
4. The key **producingStudio** indicating the name of the studio that will produce the movie using that star.

That is, the relation schema is:

**Contracts(starName, title, year, studioOfStar, producingStudio)**

Notice that we have been inventive in choosing attribute names for our relation schema, avoiding “name” for any attribute, since it would be unobvious whether that referred to a star’s name or studio’s name, and in the latter case, which studio role. Also, were there attributes attached to entity set *Contracts*, such as *salary*, these attributes would be added to the schema of relation *Contracts*.

□

#### 4.5.3 Combining Relations

Sometimes, the relations that we get from converting entity sets and relationships to relations are not the best possible choice of relations for the given data. One common situation occurs when there is an entity set *E* with a many-one relationship *R* from *E* to *F*. The relations from *E* and *R* will each have the key for *E* in their relation schema. In addition, the relation for *E* will have in its schema the attributes of *E* that are not in the key, and the relation for *R* will have the key attributes of *F* and any attributes of *R* itself. Because *R* is many-one, all these attributes are functionally determined by the key for *E*, and we can combine them into one relation with a schema consisting of:

1. All attributes of *E*.
2. The key attributes of *F*.
3. Any attributes belonging to relationship *R*.

For an entity *e* of *E* that is not related to any entity of *F*, the attributes of types (2) and (3) will have null values in the tuple for *e*.

**Example 4.28 :** In our running movie example, *Owns* is a many-one relationship from *Movies* to *Studios*, which we converted to a relation in Example 4.25. The relation obtained from entity set *Movies* was discussed in Example 4.24. We can combine these relations by taking all their attributes and forming one relation schema. If we do, the relation looks like that in Fig. 4.26. □

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>
Star Wars	1977	124	sciFi	Fox
Gone With the Wind	1939	239	drama	MGM
Wayne's World	1992	95	comedy	Paramount

Figure 4.26: Combining relation *Movies* with relation *Owns*

Whether or not we choose to combine relations in this manner is a matter of judgement. However, there are some advantages to having all the attributes that are dependent on the key of entity set  $E$  together in one relation, even if there are a number of many-one relationships from  $E$  to other entity sets. For example, it is often more efficient to answer queries involving attributes of one relation than to answer queries involving attributes of several relations. In fact, some design systems based on the E/R model combine these relations automatically.

On the other hand, one might wonder if it made sense to combine the relation for  $E$  with the relation of a relationship  $R$  that involved  $E$  but was not many-one from  $E$  to some other entity set. Doing so is risky, because it often leads to redundancy, as the next example shows.

**Example 4.29 :** To get a sense of what can go wrong, suppose we combined the relation of Fig. 4.26 with the relation that we get for the many-many relationship *Stars-in*; recall this relation was suggested by Fig. 4.24. Then the combined relation would look like Fig. 3.2, which we reproduce here as Fig. 4.27. As we discussed in Section 3.3.1, this relation has anomalies that we need to remove by the process of normalization.  $\square$

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>starName</i>
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Figure 4.27: The relation *Movies* with star information

#### 4.5.4 Handling Weak Entity Sets

When a weak entity set appears in an E/R diagram, we need to do three things differently.

1. The relation for the weak entity set  $W$  itself must include not only the attributes of  $W$  but also the key attributes of the supporting entity sets. The supporting entity sets are easily recognized because they are reached by supporting (double-diamond) relationships from  $W$ .
2. The relation for any relationship in which the weak entity set  $W$  appears must use as a key for  $W$  all of its key attributes, including those of other entity sets that contribute to  $W$ 's key.
3. However, a supporting relationship  $R$ , from the weak entity set  $W$  to a supporting entity set, need not be converted to a relation at all. The justification is that, as discussed in Section 4.5.3, the attributes of many-one relationship  $R$ 's relation will either be attributes of the relation for  $W$ , or (in the case of attributes on  $R$ ) can be added to the schema for  $W$ 's relation.

Of course, when introducing additional attributes to build the key of a weak entity set, we must be careful not to use the same name twice. If necessary, we rename some or all of these attributes.

**Example 4.30:** Let us consider the weak entity set *Crews* from Fig. 4.20, which we reproduce here as Fig. 4.28. From this diagram we get three relations, whose schemas are:

```

Studios(name, addr)
Crews(number, studioName, crewChief)
Unit-of(number, studioName, name)

```

The first relation, *Studios*, is constructed in a straightforward manner from the entity set of the same name. The second, *Crews*, comes from the weak entity set *Crews*. The attributes of this relation are the key attributes of *Crews* and the one nonkey attribute of *Crews*, which is *crewChief*. We have chosen *studioName* as the attribute in relation *Crews* that corresponds to the attribute *name* in the entity set *Studios*.

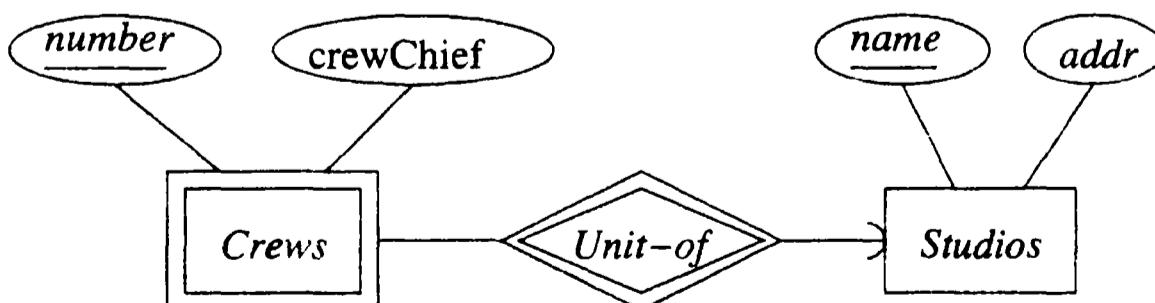


Figure 4.28: The crews example of a weak entity set

The third relation, *Unit-of*, comes from the relationship of the same name. As always, we represent an E/R relationship in the relational model by a relation whose schema has the key attributes of the related entity sets. In this case,

**Unit-of** has attributes **number** and **studioName**, the key for weak entity set *Crews*, and attribute **name**, the key for entity set *Studios*. However, notice that since *Unit-of* is a many-one relationship, the studio **studioName** is surely the same as the studio **name**.

For instance, suppose Disney crew #3 is one of the crews of the Disney studio. Then the relationship set for E/R relationship *Unit-of* includes the pair

(Disney-crew-#3, Disney)

This pair gives rise to the tuple

(3, Disney, Disney)

for the relation **Unit-of**.

Notice that, as must be the case, the components of this tuple for attributes **studioName** and **name** are identical. As a consequence, we can “merge” the attributes **studioName** and **name** of **Unit-of**, giving us the simpler schema:

**Unit-of**(**number**, **name**)

However, now we can dispense with the relation **Unit-of** altogether, since its attributes are now a subset of the attributes of relation **Crews**.  $\square$

The phenomenon observed in Example 4.30 — that a supporting relationship needs no relation — is universal for weak entity sets. The following is a modified rule for converting to relations entity sets that are weak.

- If  $W$  is a weak entity set, construct for  $W$  a relation whose schema consists of:
  1. All attributes of  $W$ .
  2. All attributes of supporting relationships for  $W$ .
  3. For each supporting relationship for  $W$ , say a many-one relationship from  $W$  to entity set  $E$ , all the *key* attributes of  $E$ .

Rename attributes, if necessary, to avoid name conflicts.

- Do *not* construct a relation for any supporting relationship for  $W$ .

#### 4.5.5 Exercises for Section 4.5

**Exercise 4.5.1:** Convert the E/R diagram of Fig. 4.29 to a relational database schema.

! **Exercise 4.5.2:** There is another E/R diagram that could describe the weak entity set *Bookings* in Fig. 4.29. Notice that a booking can be identified uniquely by the flight number, day of the flight, the row, and the seat; the customer is not then necessary to help identify the booking.

## Relations With Subset Schemas

You might imagine from Example 4.30 that whenever one relation  $R$  has a set of attributes that is a subset of the attributes of another relation  $S$ , we can eliminate  $R$ . That is not exactly true.  $R$  might hold information that doesn't appear in  $S$  because the additional attributes of  $S$  do not allow us to extend a tuple from  $R$  to  $S$ .

For instance, the Internal Revenue Service tries to maintain a relation **People**(name, ss#) of potential taxpayers and their social-security numbers, even if the person had no income and did not file a tax return. They might also maintain a relation **TaxPayers**(name, ss#, amount) indicating the amount of tax paid by each person who filed a return in the current year. The schema of **People** is a subset of the schema of **TaxPayers**, yet there may be value in remembering the social-security number of those who are mentioned in **People** but not in **Taxpayers**.

In fact, even identical sets of attributes may have different semantics, so it is not possible to merge their tuples. An example would be two relations **Stars**(name, addr) and **Studios**(name, addr). Although the schemas look alike, we cannot turn star tuples into studio tuples, or vice-versa.

On the other hand, when the two relations come from the weak-entity-set construction, then there can be no such additional value to the relation with the smaller set of attributes. The reason is that the tuples of the relation that comes from the supporting relationship correspond one-for-one with the tuples of the relation that comes from the weak entity set. Thus, we routinely eliminate the former relation.

- a) Revise the diagram of Fig. 4.29 to reflect this new viewpoint.
- b) Convert your diagram from (a) into relations. Do you get the same database schema as in Exercise 4.5.1?

**Exercise 4.5.3:** The E/R diagram of Fig. 4.30 represents ships. Ships are said to be *sisters* if they were designed from the same plans. Convert this diagram to a relational database schema.

**Exercise 4.5.4:** Convert the following E/R diagrams to relational database schemas.

- a) Figure 4.22.
- b) Your answer to Exercise 4.4.1.
- c) Your answer to Exercise 4.4.4(a).
- d) Your answer to Exercise 4.4.4(b).

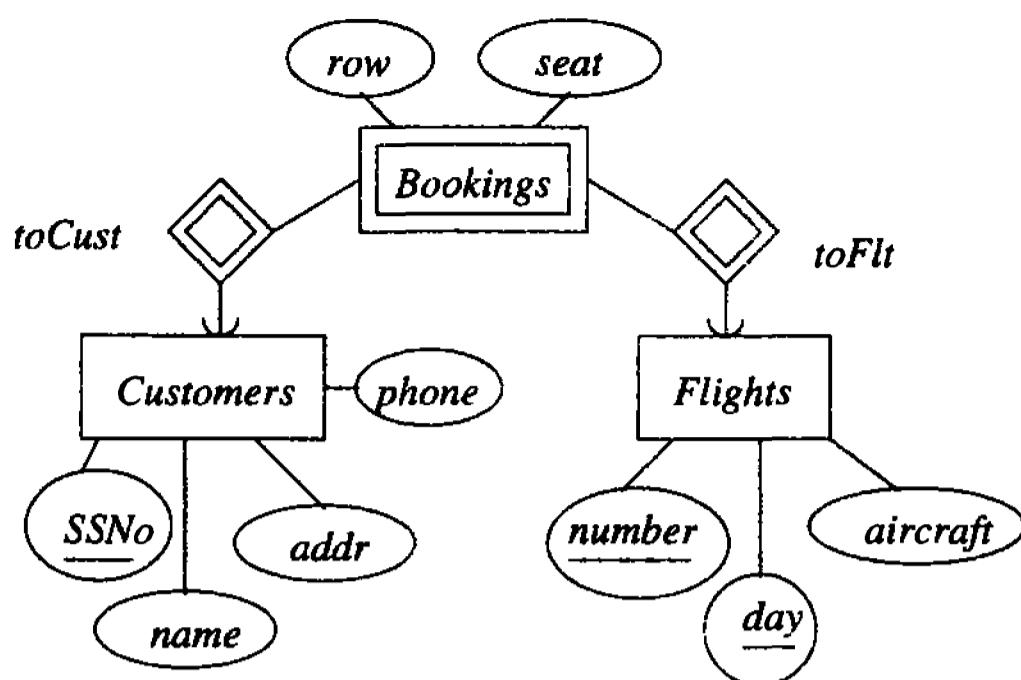


Figure 4.29: An E/R diagram about airlines

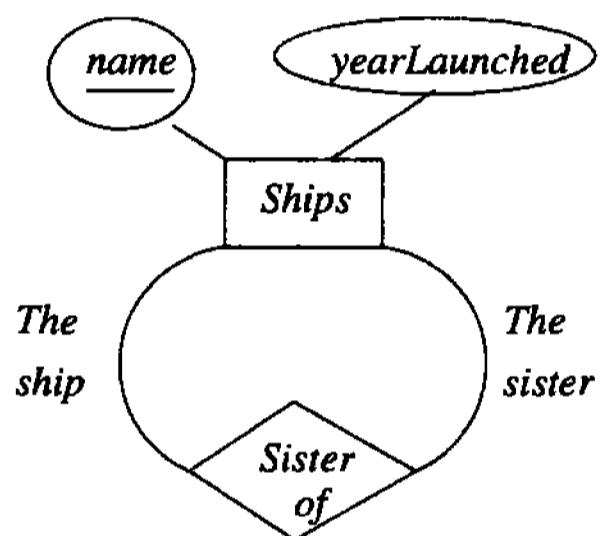


Figure 4.30: An E/R diagram about sister ships

## 4.6 Converting Subclass Structures to Relations

When we have an *isa-hierarchy* of entity sets, we are presented with several choices of strategy for conversion to relations. Recall we assume that:

- There is a root entity set for the hierarchy,
- This entity set has a key that serves to identify every entity represented by the hierarchy, and
- A given entity may have *components* that belong to the entity sets of any subtree of the hierarchy, as long as that subtree includes the root.

The principal conversion strategies are:

1. *Follow the E/R viewpoint.* For each entity set  $E$  in the hierarchy, create a relation that includes the key attributes from the root and any attributes belonging to  $E$ .

2. *Treat entities as objects belonging to a single class.* For each possible subtree that includes the root, create one relation, whose schema includes all the attributes of all the entity sets in the subtree.
3. *Use null values.* Create one relation with all the attributes of all the entity sets in the hierarchy. Each entity is represented by one tuple, and that tuple has a null value for whatever attributes the entity does not have.

We shall consider each approach in turn.

#### 4.6.1 E/R-Style Conversion

Our first approach is to create a relation for each entity set, as usual. If the entity set  $E$  is not the root of the hierarchy, then the relation for  $E$  will include the key attributes at the root, to identify the entity represented by each tuple, plus all the attributes of  $E$ . In addition, if  $E$  is involved in a relationship, then we use these key attributes to identify entities of  $E$  in the relation corresponding to that relationship.

Note, however, that although we spoke of “isa” as a relationship, it is unlike other relationships, in that it connects components of a single entity, not distinct entities. Thus, we do not create a relation for “isa.”

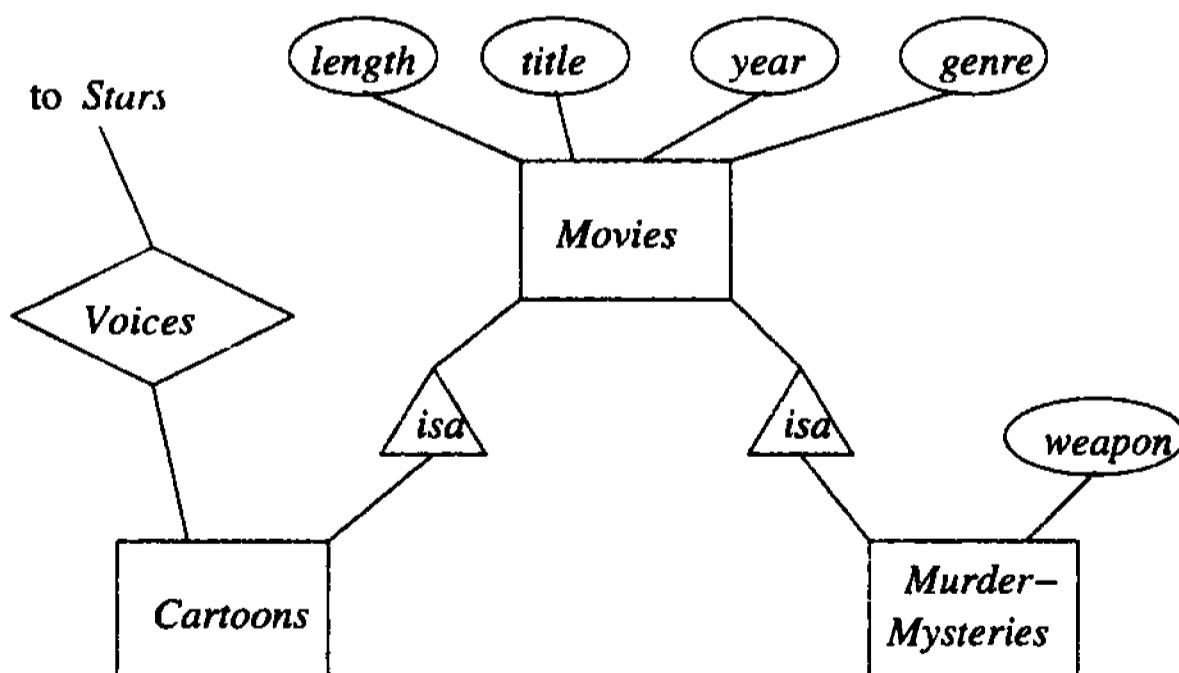


Figure 4.31: The movie hierarchy

**Example 4.31:** Consider the hierarchy of Fig. 4.10, which we reproduce here as Fig. 4.31. The relations needed to represent the entity sets in this hierarchy are:

1. **Movies(title, year, length, genre).** This relation was discussed in Example 4.24, and every movie is represented by a tuple here.

2. **MurderMysteries**(*title*, *year*, *weapon*). The first two attributes are the key for all movies, and the last is the lone attribute for the corresponding entity set. Those movies that are murder mysteries have a tuple here as well as in **Movies**.
3. **Cartoons**(*title*, *year*). This relation is the set of cartoons. It has no attributes other than the key for movies, since the extra information about cartoons is contained in the relationship *Voices*. Movies that are cartoons have a tuple here as well as in **Movies**.

Note that the fourth kind of movie — those that are both cartoons and murder mysteries — have tuples in all three relations.

In addition, we shall need the relation **Voices**(*title*, *year*, *starName*) that corresponds to the relationship *Voices* between *Stars* and *Cartoons*. The last attribute is the key for *Stars* and the first two form the key for *Cartoons*.

For instance, the movie *Roger Rabbit* would have tuples in all four relations. Its basic information would be in **Movies**, the murder weapon would appear in **MurderMysteries**, and the stars that provided voices for the movie would appear in **Voices**.

Notice that the relation **Cartoons** has a schema that is a subset of the schema for the relation **Voices**. In many situations, we would be content to eliminate a relation such as **Cartoons**, since it appears not to contain any information beyond what is in **Voices**. However, there may be silent cartoons in our database. Those cartoons would have no voices, and we would therefore lose information should we eliminate relation **Cartoons**.  $\square$

#### 4.6.2 An Object-Oriented Approach

An alternative strategy for converting isa-hierarchies to relations is to enumerate all the possible subtrees of the hierarchy. For each, create one relation that represents entities having components in exactly those subtrees. The schema for this relation has all the attributes of any entity set in the subtree. We refer to this approach as “object-oriented,” since it is motivated by the assumption that entities are “objects” that belong to one and only one class.

**Example 4.32:** Consider the hierarchy of Fig. 4.31. There are four possible subtrees including the root:

1. *Movies* alone.
2. *Movies* and *Cartoons* only.
3. *Movies* and *Murder-Mysteries* only.
4. All three entity sets.

We must construct relations for all four “classes.” Since only *Murder-Mysteries* contributes an attribute that is unique to its entities, there is actually some repetition, and these four relations are:

```
Movies(title, year, length, genre)
MoviesC(title, year, length, genre)
MoviesMM(title, year, length, genre, weapon)
MoviesCMM(title, year, length, genre, weapon)
```

If *Cartoons* had attributes unique to that entity set, then all four relations would have different sets of attributes. As that is not the case here, we could combine **Movies** with **MoviesC** (i.e., create one relation for non-murder-mysteries) and combine **MoviesMM** with **MoviesCMM** (i.e., create one relation for all murder mysteries), although doing so loses some information — which movies are cartoons.

We also need to consider how to handle the relationship *Voices* from *Cartoons* to *Stars*. If *Voices* were many-one from *Cartoons*, then we could add a *voice* attribute to **MoviesC** and **MoviesCMM**, which would represent the *Voices* relationship and would have the side-effect of making all four relations different. However, *Voices* is many-many, so we need to create a separate relation for this relationship. As always, its schema has the key attributes from the entity sets connected; in this case

```
Voices(title, year, starName)
```

would be an appropriate schema.

One might consider whether it was necessary to create two such relations, one connecting cartoons that are not murder mysteries to their voices, and the other for cartoons that *are* murder mysteries. However, there does not appear to be any benefit to doing so in this case. □

#### 4.6.3 Using Null Values to Combine Relations

There is one more approach to representing information about a hierarchy of entity sets. If we are allowed to use **NONE** (the null value as in SQL) as a value in tuples, we can handle a hierarchy of entity sets with a single relation. This relation has all the attributes belonging to any entity set of the hierarchy. An entity is then represented by a single tuple. This tuple has **NONE** in each attribute that is not defined for that entity.

**Example 4.33:** If we applied this approach to the diagram of Fig. 4.31, we would create a single relation whose schema is:

```
Movie(title, year, length, genre, weapon)
```

Those movies that are not murder mysteries would have **NONE** in the **weapon** component of their tuple. It would also be necessary to have a relation *Voices* to connect those movies that are cartoons to the stars performing the voices, as in Example 4.32. □

#### 4.6.4 Comparison of Approaches

Each of the three approaches, which we shall refer to as “straight-E/R,” “object-oriented,” and “nulls,” respectively, have advantages and disadvantages. Here is a list of the principal issues.

1. It can be expensive to answer queries involving several relations, so we would prefer to find all the attributes we needed to answer a query in one relation. The nulls approach uses only one relation for all the attributes, so it has an advantage in this regard. The other two approaches have advantages for different kinds of queries. For instance:
  - (a) A query like “what films of 2008 were longer than 150 minutes?” can be answered directly from the relation **Movies** in the straight-E/R approach of Example 4.31. However, in the object-oriented approach of Example 4.32, we need to examine **Movies**, **MoviesC**, **MoviesMM**, and **MoviesCMM**, since a long movie may be in any of these four relations.
  - (b) On the other hand, a query like “what weapons were used in cartoons of over 150 minutes in length?” gives us trouble in the straight-E/R approach. We must access **Movies** to find those movies of over 150 minutes. We must access **Cartoons** to verify that a movie is a cartoon, and we must access **MurderMysteries** to find the murder weapon. In the object-oriented approach, we have only to access the relation **MoviesCMM**, where all the information we need will be found.
2. We would like not to use too many relations. Here again, the nulls method shines, since it requires only one relation. However, there is a difference between the other two methods, since in the straight-E/R approach, we use only one relation per entity set in the hierarchy. In the object-oriented approach, if we have a root and  $n$  children ( $n + 1$  entity sets in all), then there are  $2^n$  different classes of entities, and we need that many relations.
3. We would like to minimize space and avoid repeating information. Since the object-oriented method uses only one tuple per entity, and that tuple has components for only those attributes that make sense for the entity, this approach offers the minimum possible space usage. The nulls approach also has only one tuple per entity, but these tuples are “long”; i.e., they have components for all attributes, whether or not they are appropriate for a given entity. If there are many entity sets in the hierarchy, and there are many attributes among those entity sets, then a large fraction of the space could be wasted in the nulls approach. The straight-E/R method has several tuples for each entity, but only the key attributes are repeated. Thus, this method could use either more or less space than the nulls method.

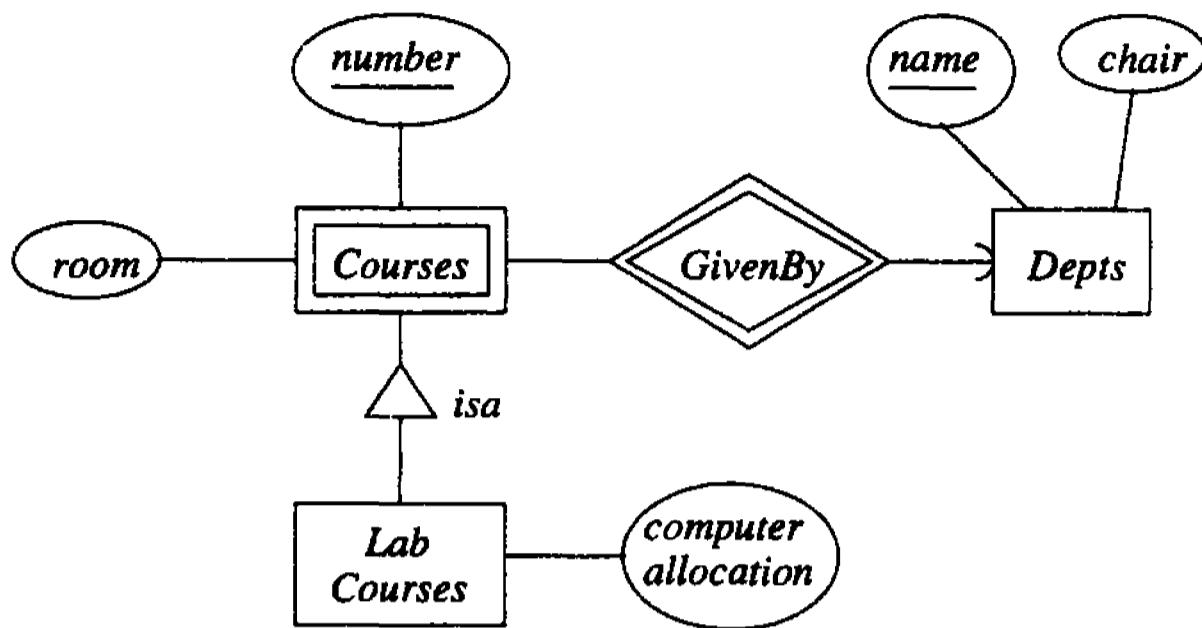


Figure 4.32: E/R diagram for Exercise 4.6.1

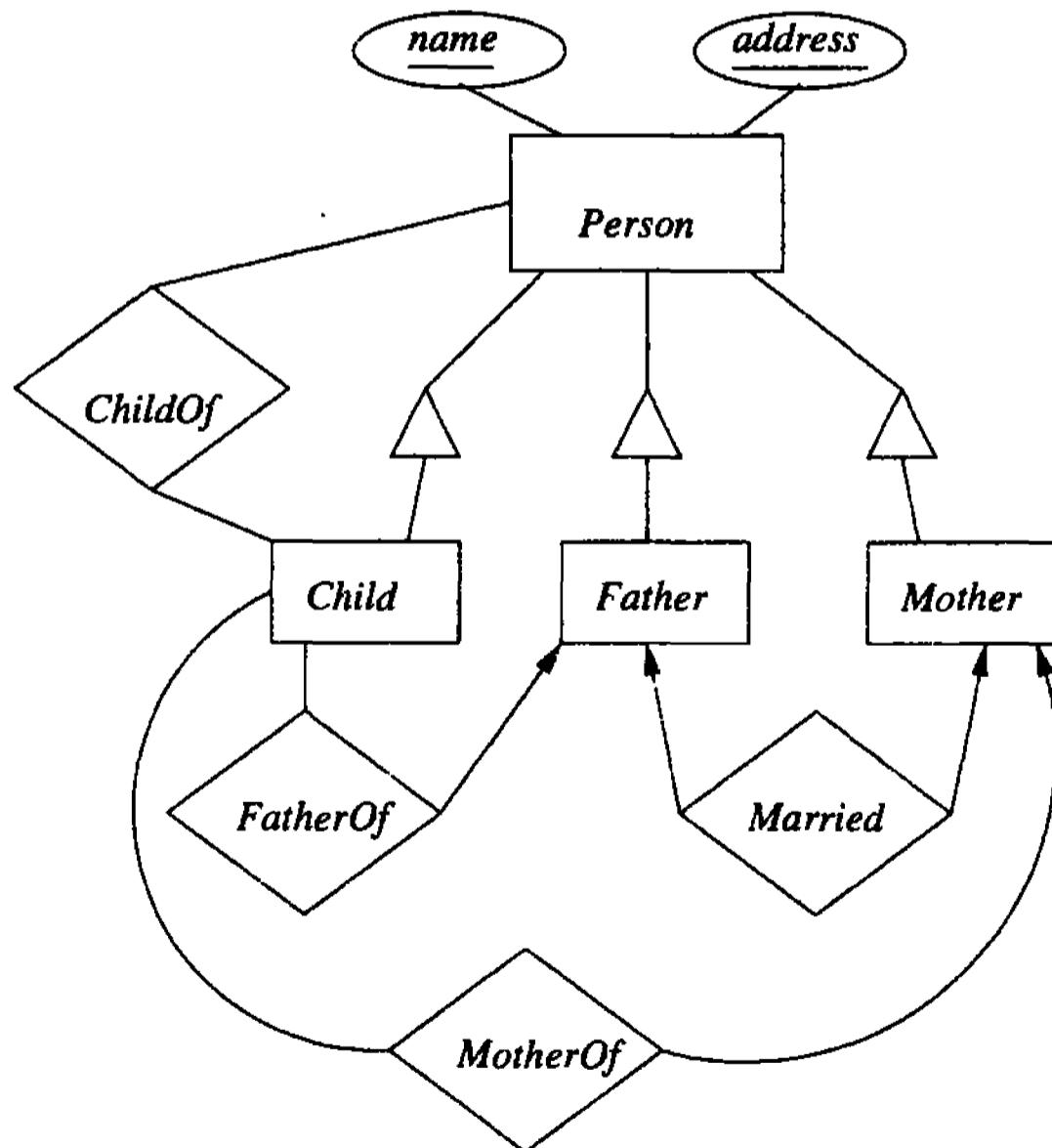


Figure 4.33: E/R diagram for Exercise 4.6.2

### 4.6.5 Exercises for Section 4.6

**Exercise 4.6.1:** Convert the E/R diagram of Fig. 4.32 to a relational database schema, using each of the following approaches:

- a) The straight-E/R method.
- b) The object-oriented method.
- c) The nulls method.

**! Exercise 4.6.2:** Convert the E/R diagram of Fig. 4.33 to a relational database schema, using:

- a) The straight-E/R method.
- b) The object-oriented method.
- c) The nulls method.

**Exercise 4.6.3:** Convert your E/R design from Exercise 4.1.7 to a relational database schema, using:

- a) The straight-E/R method.
- b) The object-oriented method.
- c) The nulls method.

**! Exercise 4.6.4:** Suppose that we have an *isa*-hierarchy involving  $e$  entity sets. Each entity set has  $a$  attributes, and  $k$  of those at the root form the key for all these entity sets. Give formulas for (i) the minimum and maximum number of relations used, and (ii) the minimum and maximum number of components that the tuple(s) for a single entity have all together, when the method of conversion to relations is:

- a) The straight-E/R method.
- b) The object-oriented method.
- c) The nulls method.

## 4.7 Unified Modeling Language

UML (*Unified Modeling Language*) was developed originally as a graphical notation for describing software designs in an object-oriented style. It has been extended, with some modifications, to be a popular notation for describing database designs, and it is this portion of UML that we shall study here. UML offers much the same capabilities as the E/R model, with the exception of multiway relationships. UML also offers the ability to treat entity sets as true classes, with methods as well as data. Figure 4.34 summarizes the common concepts, with different terminology, used by E/R and UML.

UML	E/R Model
Class	Entity set
Association	Binary relationship
Association Class	Attributes on a relationship
Subclass	Isa hierarchy
Aggregation	Many-one relationship
Composition	Many-one relationship with referential integrity

Figure 4.34: Comparison between UML and E/R terminology

#### 4.7.1 UML Classes

A class in UML is similar to an entity set in the E/R model. The notation for a class is rather different, however. Figure 4.35 shows the class that corresponds to the E/R entity set *Movies* from our running example of this chapter.

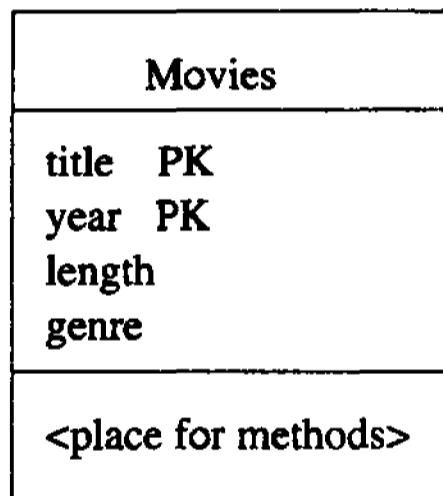


Figure 4.35: The *Movies* class in UML

The box for a class is divided into three parts. At the top is the name of the class. The middle has the attributes, which are like instance variables of a class. In our *Movies* class, we use the attributes *title*, *year*, *length*, and *genre*.

The bottom portion is for methods. Neither the E/R model nor the relational model provides methods. However, they are an important concept, and one that actually appears in modern relational systems, called “object-relational” DBMS’s (see Section 10.3).

**Example 4.34:** We might have added an instance method *lengthInHours()*. The UML specification doesn’t tell anything more about a method than the types of any arguments and the type of its return-value. Perhaps this method returns *length/60.0*, but we cannot know from the design. □

In this section, we shall not use methods in our design. Thus, in the future, UML class boxes will have only two sections, for the class name and the attributes.

### 4.7.2 Keys for UML classes

As for entity sets, we can declare one key for a UML class. To do so, we follow each attribute in the key by the letters PK, standing for “primary key.” There is no convenient way to stipulate that several attributes or sets of attributes are each keys.

**Example 4.35 :** In Fig. 4.35, we have made our standard assumption that *title* and *year* together form the key for *Movies*. Notice that PK appears on the lines for these attributes and not for the others. □

### 4.7.3 Associations

A binary relationship between classes is called an *association*. There is no analog of multiway relationships in UML. Rather, a multiway relationship has to be broken into binary relationships, which as we suggested in Section 4.1.10, can always be done. The interpretation of an association is exactly what we described for relationships in Section 4.1.5 on relationship sets. The association is a set of pairs of objects, one from each of the classes it connects.

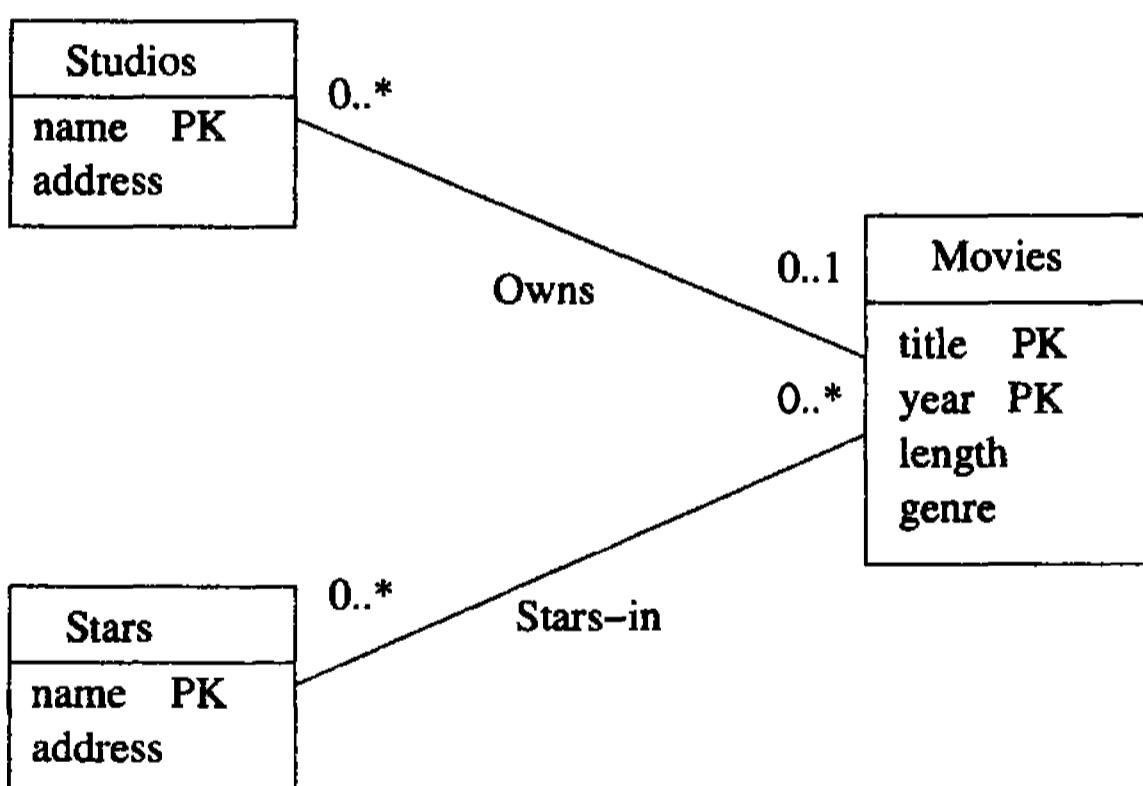


Figure 4.36: Movies, stars, and studios in UML

We draw a UML association between two classes simply by drawing a line between them, and giving the line a name. Usually, we'll place the name below the line. For example, Fig. 4.36 is the UML analog of the E/R diagram of Fig. 4.17. There are two associations, *Stars-in* and *Owns*; the first connects *Movies* with *Stars* and the second connects *Movies* with *Studios*.

Every association has constraints on the number of objects from each of its classes that can be connected to an object of the other class. We indicate these constraints by a label of the form  $m..n$  at each end. The meaning of this label is that each object at the other end is connected to at least  $m$  and at most  $n$  objects at this end. In addition:

- A \* in place of  $n$ , as in  $m..*$ , stands for “infinity.” That is, there is no upper limit.
- A \* alone, in place of  $m..n$ , stands for the range  $0..*$ , that is, no constraint at all on the number of objects.
- If there is no label at all at an end of an association edge, then the label is taken to be  $1..1$ , i.e., “exactly one.”

**Example 4.36:** In Fig. 4.36 we see  $0..*$  at the *Movies* end of both associations. That says that a star appears in zero or more movies, and a studio owns zero or more movies; i.e., there is no constraint for either. There is also a  $0..*$  at the *Stars* end of association *Stars-in*, telling us that a movie has any number of stars. However, the label on the *Studios* end of association *Owns* is  $0..1$ , which means either 0 or 1 studio. That is, a given movie can either be owned by one studio, or not be owned by any studio in the database. Notice that this constraint is exactly what is said by the pointed arrow entering *Studios* in the E/R diagram of Fig. 4.17. □

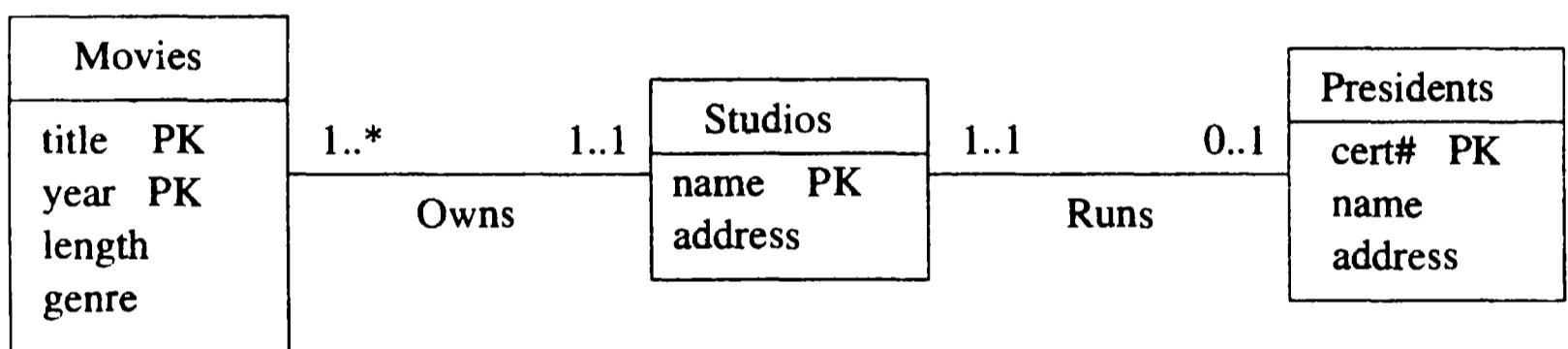


Figure 4.37: Expressing referential integrity in UML

**Example 4.37:** The UML diagram of Fig. 4.37 is intended to mirror the E/R diagram of Fig. 4.18. Here, we see assumptions somewhat different from those in Example 4.36, about the numbers of movies and studios that can be associated. The label  $1..*$  at the *Movies* end of *Owns* says that each studio must own at least one movie (or else it isn't really a studio). There is still no upper limit on how many movies a studio can own.

At the *Studios* end of *Owns*, we see the label  $1..1$ . That label says that a movie must be owned by one studio and only one studio. It is not possible for a movie not to be owned by any studio, as was possible in Fig. 4.36. The label  $1..1$  says exactly what the rounded arrow in E/R diagrams says.

We also see the association *Runs* between studios and presidents. At the *Studios* end we see label  $1..1$ . That is, a president must be the president of one and only one studio. That label reflects the same constraint as the rounded arrow from *Presidents* to *Studios* in Fig. 4.18. At the other end of association *Runs* is the label  $0..1$ . That label says that a studio can have at most one president, but it could not have a president at some time. This constraint is exactly the constraint of a pointed arrow. □

### 4.7.4 Self-Associations

An association can have both ends at the same class; such an association is called a *self-association*. To distinguish the two roles played by one class in a self-association, we give the association two names, one for each end.

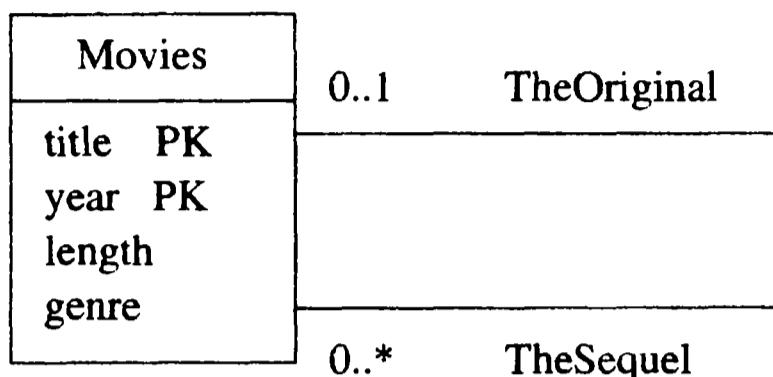


Figure 4.38: A self-association representing sequels of movies

**Example 4.38:** Figure 4.38 represents the relationship “sequel-of” on movies. We see one association with each end at the class *Movies*. The end with role *TheOriginal* points to the original movie, and it has label 0..1. That is, for a movie to be a sequel, there has to be exactly one movie that was the original. However, some movies are not sequels of any movie. The other role, *TheSequel* has label 0..\*. The reasoning is that an original can have any number of sequels. Note we take the point of view that there is an original movie for any sequence of sequels, and a sequel is a sequel of the original, not of the previous movie in the sequence. For instance, *Rocky II* through *Rocky V* are sequels of *Rocky*. We do not assume *Rocky IV* is a sequel of *Rocky III*, and so on. □

### 4.7.5 Association Classes

We can attach attributes to an association in much the way we did in the E/R model, in Section 4.1.9.<sup>5</sup> In UML, we create a new class, called an *association class*, and attach it to the middle of the association. The association class has its own name, but its attributes may be thought of as attributes of the association to which it attaches.

**Example 4.39:** Suppose we want to add to the association *Stars-in* between *Movies* and *Stars* some information about the compensation the star received for the movie. This information is not associated with the movie (different stars get different salaries) nor with the star (stars can get different salaries for different movies). Thus, we must attach this information with the association itself. That is, every movie-star pair has its own salary information.

Figure 4.39 shows the association *Stars-in* with an association class called *Compensation*. This class has two attributes, *salary* and *residuals*. Notice

<sup>5</sup>However, the example there in Fig. 4.7 will not carry over directly, because the relationship there is 3-way.

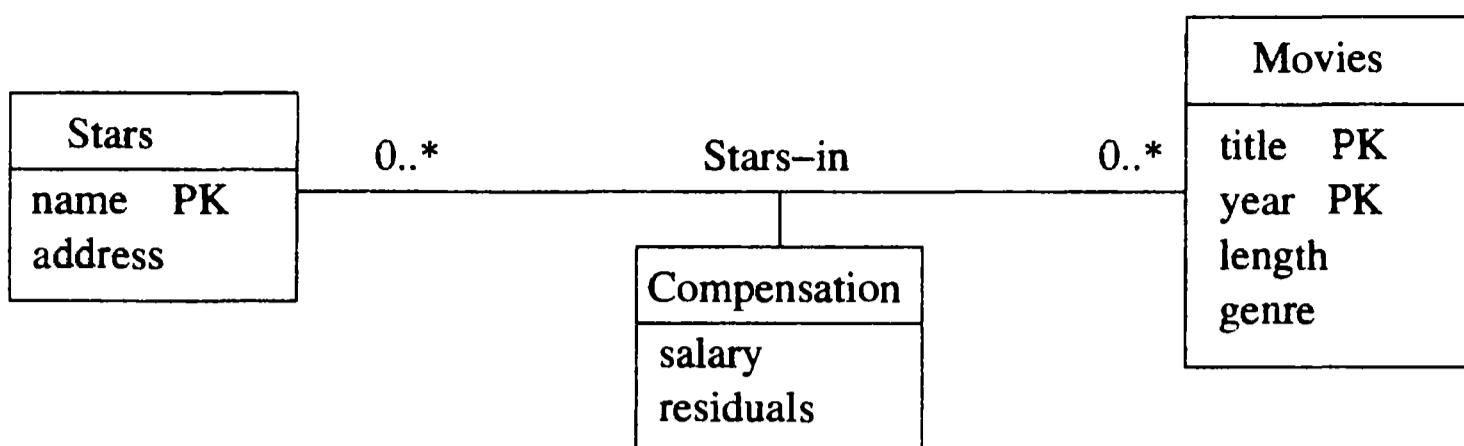


Figure 4.39: *Compensation* is an association class for the association *Stars-in*

that there is no primary key marked for *Compensation*. When we convert a diagram such as Fig. 4.39 to relations, the attributes of *Compensation* will attach to tuples created for movie-star pairs, as was described for relationships in Section 4.5.2. □

#### 4.7.6 Subclasses in UML

Any UML class can have a hierarchy of subclasses below it. The primary key comes from the root of the hierarchy, just as with E/R hierarchies. UML permits a class  $C$  to have four different kinds of subclasses below it, depending on our choices of answer to two questions:

1. *Complete versus Partial*. Is every object in the class  $C$  a member of some subclass? If so, the subclasses are *complete*; otherwise they are *partial* or *incomplete*.
2. *Disjoint versus Overlapping*. Are the subclasses *disjoint* (an object cannot be in two of the subclasses)? If an object can be in two or more of the subclasses, then the subclasses are said to be *overlapping*.

Note that these decisions are taken at each level of a hierarchy, and the decisions may be made independently at each point.

There are several interesting relationships between the classification of UML subclasses given above, the standard notion of subclasses in object-oriented systems, and the E/R notion of subclasses.

- In a typical object-oriented system, subclasses are disjoint. That is, no object can be in two classes. Of course they inherit properties from their parent class, so in a sense, an object also “belongs” in the parent class. However, the object may not also be in a sibling class.
- The E/R model automatically allows overlapping subclasses.
- Both the E/R model and object-oriented systems allow either complete or partial subclasses. That is, there is no requirement that a member of the superclass be in any subclass.

Subclasses are represented by rectangles, like any class. We assume a subclass inherits the properties (attributes and associations) from its superclass. However, any additional attributes belonging to the subclass are shown in the box for that subclass, and the subclass may have its own, additional, associations to other classes. To represent the class/subclass relationship in UML diagrams, we use a triangular, open arrow pointing to the superclass. The subclasses are usually connected by a horizontal line, feeding into the arrow.

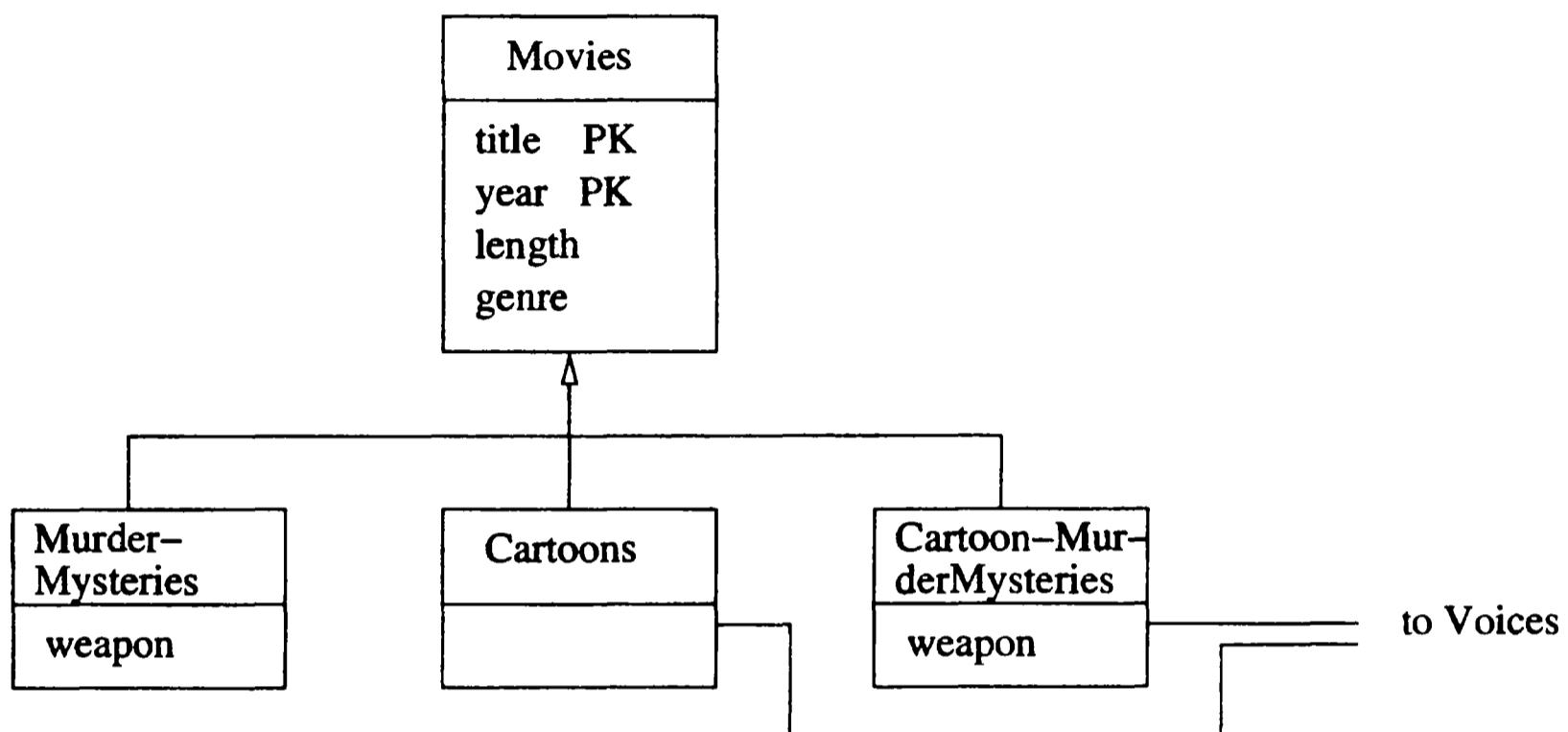


Figure 4.40: Cartoons and murder mysteries as disjoint subclasses of movies

**Example 4.40:** Figure 4.40 shows a UML variant of the subclass example from Section 4.1.11. However, unlike the E/R subclasses, which are of necessity overlapping, we have chosen here to make the subclasses disjoint. They are partial, of course, since many movies are neither cartoons nor murder mysteries.

Because the subclasses were chosen disjoint, there must be a third subclass for movies like *Roger Rabbit* that are both cartoons and murder mysteries. Notice that both the classes *MurderMysteries* and *Cartoon-MurderMysteries* have additional attribute *weapon*, while the two subclasses *MurderMysteries* and *Cartoon-MurderMysteries* have associations with the unseen class *Voices*.

□

#### 4.7.7 Aggregations and Compositions

There are two special notations for many-one associations whose implications are rather subtle. In one sense, they reflect the object-oriented style of programming, where it is common for one class to have references to other classes among its attributes. In another sense, these special notations are really stipulations about how the diagram should be converted to relations; we discuss this aspect of the matter in Section 4.8.3.

An *aggregation* is a line between two classes that ends in an open diamond at one end. The implication of the diamond is that the label at that end must

be 0..1, i.e., the aggregation is a many-one association from the class at the opposite end to the class at the diamond end. Although the aggregation is an association, we do not need to name it, since in practice that name will never be used in a relational implementation.

A *composition* is similar to an association, but the label at the diamond end must be 1..1. That is, every object at the opposite end from the diamond must be connected to exactly one object at the diamond end. Compositions are distinguished by making the diamond be solid black.

**Example 4.41:** In Fig. 4.41 we see examples of both an aggregation and a composition. It both modifies and elaborates on the situation of Fig. 4.37. We see an association from *Movies* to *Studios*. The label 1..\* at the *Movies* end says that a studio has to own at least one movie. We do not need a label at the diamond end, since the open diamond implies a 0..1 label. That is, a movie may or may not be associated with a studio, but cannot be associated with more than one studio. There is also the implication that *Movies* objects will contain a reference to their owning *Studios* object; that reference may be null if the movie is not owned by a studio.

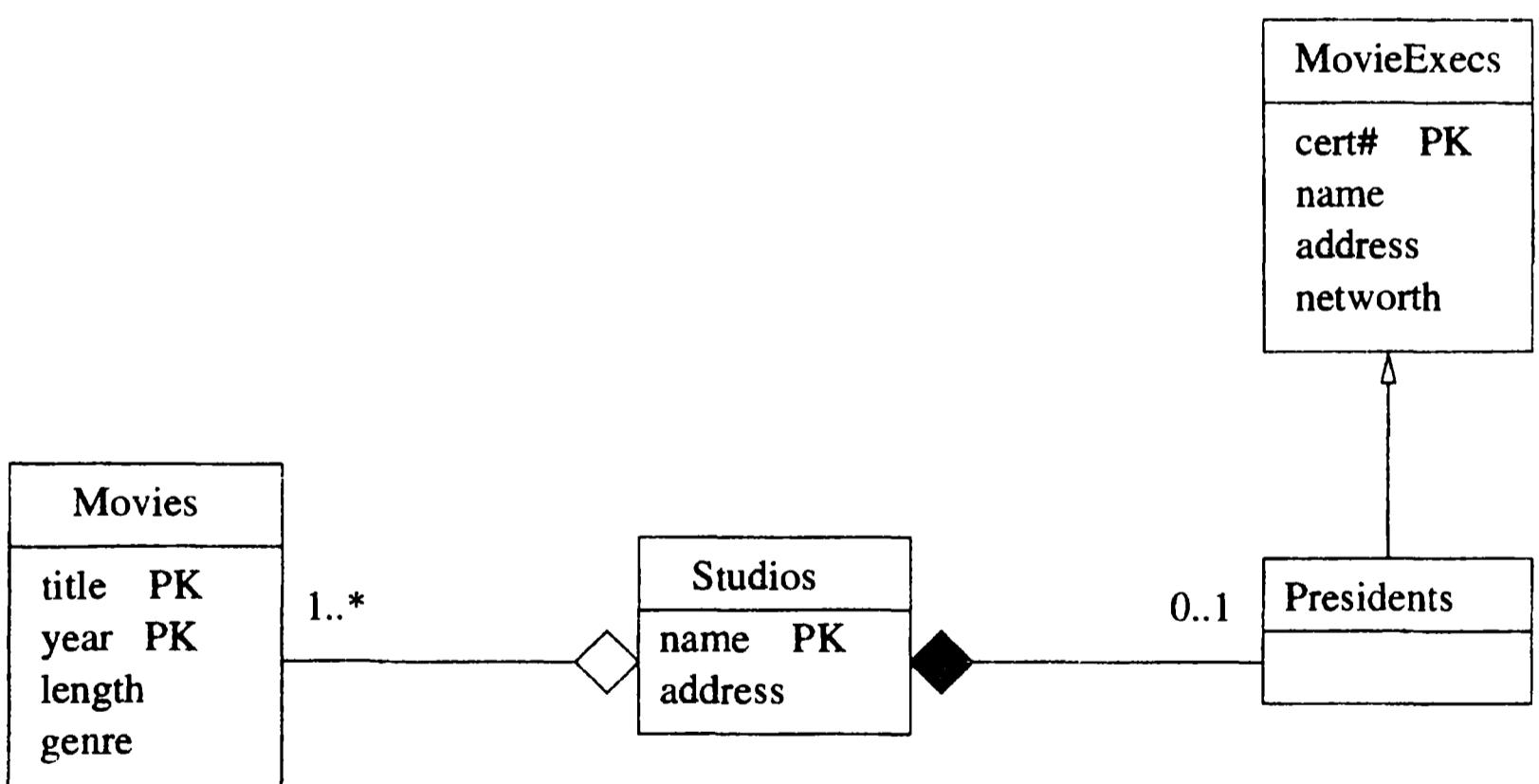


Figure 4.41: An aggregation from *Movies* to *Studios* and a composition from *Presidents* to *Studios*

At the right, we see the class *MovieExecs* with a subclass *Presidents*. There is a composition from *Presidents* to *Studios*, meaning that every president is the president of exactly one studio. A label 1..1 at the *Studios* end is implied by the solid diamond. The implication of the composition is that *Presidents* objects will contain a reference to a *Studios* object, and that this reference cannot be null. □

### 4.7.8 Exercises for Section 4.7

**Exercise 4.7.1:** Draw a UML diagram for the problem of Exercise 4.1.1.

**Exercise 4.7.2:** Modify your diagram from Exercise 4.7.1 in accordance with the requirements of Exercise 4.1.2.

**Exercise 4.7.3:** Repeat Exercise 4.1.3 using UML.

**Exercise 4.7.4:** Repeat Exercise 4.1.6 using UML.

**Exercise 4.7.5:** Repeat Exercise 4.1.7 using UML. Are your subclasses disjoint or overlapping? Are they complete or partial?

**Exercise 4.7.6:** Repeat Exercise 4.1.9 using UML.

**Exercise 4.7.7:** Convert the E/R diagram of Fig. 4.30 to a UML diagram.

! **Exercise 4.7.8:** How would you represent the 3-way relationship of *Contracts* among movies, stars, and studios (see Fig. 4.4) in UML?

! **Exercise 4.7.9:** Repeat Exercise 4.2.5 using UML.

**Exercise 4.7.10:** Usually, when we constrain associations with a label of the form  $m..n$ , we find that  $m$  and  $n$  are each either 0, 1, or \*. Give some examples of associations where it would make sense for at least one of  $m$  and  $n$  to be something different.

## 4.8 From UML Diagrams to Relations

Many of the ideas needed to turn E/R diagrams into relations work for UML diagrams as well. We shall therefore briefly review the important techniques, dwelling only on points where the two modeling methods diverge.

### 4.8.1 UML-to-Relations Basics

Here is an outline of the points that should be familiar from our discussion in Section 4.5:

- *Classes to Relations.* For each class, create a relation whose name is the name of the class, and whose attributes are the attributes of the class.
- *Associations to Relations.* For each association, create a relation with the name of that association. The attributes of the relation are the key attributes of the two connected classes. If there is a coincidence of attributes between the two classes, rename them appropriately. If there is an association class attached to the association, include the attributes of the association class among the attributes of the relation.

**Example 4.42 :** Consider the UML diagram of Fig. 4.36. For the three classes we create relations:

```
Movies(title, year, length genre)
Stars(name, address)
Studios(name, address)
```

For the two associations, we create relations

```
Stars-In(movieTitle, movieYear, starName)
Owns(movieTitle, movieYear, studioName)
```

Note that we have taken some liberties with the names of attributes, for clarity of intention, even though we were not required to do so.

For another example, consider the UML diagram of Fig. 4.39, which shows an association class. The relations for the classes *Movies* and *Stars* would be the same as above. However, for the association, we would have a relation

```
Stars-In(movieTitle, movieYear, starName, salary, residuals)
```

That is, we add to the key attributes of the associated classes, the two attributes of the association class *Compensation*. Note that there is no relation created for *Compensation* itself. □

#### 4.8.2 From UML Subclasses to Relations

The three options we enumerated in Section 4.6 apply to UML subclass hierarchies as well. Recall these options are “E/R style” (relations for each subclass have only the key attributes and attributes of that subclass), “object-oriented” (each entity is represented in the relation for only one subclass), and “use nulls” (one relation for all subclasses). However, if we have information about whether subclasses are disjoint or overlapping, and complete or partial, then we may find one or another method more appropriate. Here are some considerations:

1. If a hierarchy is disjoint at every level, then an object-oriented representation is suggested. We do not have to consider each possible tree of subclasses when forming relations, since we know that each object can belong to only one class and its ancestors in the hierarchy. Thus, there is no possibility of an exponentially exploding number of relations being created.
2. If the hierarchy is both complete and disjoint at every level, then the task is even simpler. If we use the object-oriented approach, then we have only to construct relations for the classes at the leaves of the hierarchy.
3. If the hierarchy is large and overlapping at some or all levels, then the E/R approach is indicated. We are likely to need so many relations that the relational database schema becomes unwieldy.

### 4.8.3 From Aggregations and Compositions to Relations

Aggregations and compositions are really types of many-one associations. Thus, one approach to their representation in a relational database schema is to convert them as we do for any association in Section 4.8.1. Since these elements are not necessarily named in the UML diagram, we need to invent a name for the corresponding relation.

However, there is a hidden assumption that this implementation of aggregations and compositions is undesirable. Recall from Section 4.5.3 that when we have an entity set  $E$  and a many-one relationship  $R$  from  $E$  to another entity set  $F$ , we have the option — some would say the obligation — to combine the relation for  $E$  with the relation for  $R$ . That is, the one relation constructed from  $E$  and  $R$  has all the attributes of  $E$  plus the key attributes of  $F$ .

We suggest that aggregations and compositions be treated routinely in this manner. Construct no relation for the aggregation or composition. Rather, add to the relation for the class at the nondiamond end the key attribute(s) of the class at the diamond end. In the case of an aggregation (but not a composition), it is possible that these attributes can be null.

**Example 4.43:** Consider the UML diagram of Fig. 4.41. Since there is a small hierarchy, we need to decide how *MovieExecs* and *Presidents* will be translated. Let us adopt the E/R approach, so the *Presidents* relation has only the `cert#` attribute from *MovieExecs*.

The aggregation from *Movies* to *Studios* is represented by putting the key *name* for *Studios* among the attributes for the relation *Movies*. The composition from *Presidents* to *Studios* is represented by adding the key for *Studios* to the relation *Presidents* as well. No relations are constructed for the aggregation or the composition. The following are all the relations we construct from this UML diagram.

```
MovieExecs(cert#, name, address, netWorth)
Presidents(cert#, studioName)
Movies(title, year, length, genre, studioName)
Studios(name, address)
```

As before, we take some liberties with names of attributes to make our intentions clear. □

### 4.8.4 The UML Analog of Weak Entity Sets

We have not mentioned a UML notation that corresponds to the double-border notation for weak entity sets in the E/R model. There is a sense in which none is needed. The reason is that UML, unlike E/R, draws on the tradition of object-oriented systems, which takes the point of view that each object has its own *object-identity*. That is, we can distinguish two objects, even if they have the same values for each of their attributes and other properties. That object-identity is typically viewed as a reference or pointer to the object.

In UML, we can take the point of view that the objects belonging to a class likewise have object-identity. Thus, even if the stated attributes for a class do not serve to identify a unique object of the class, we can create a new attribute that serves as a key for the corresponding relation and represents the object-identity of the object.

However, it is also possible, in UML, to use a composition as we used supporting relationships for weak entity sets in the E/R model. This composition goes from the “weak” class (the class whose attributes do not provide its key) to the “supporting” class. If there are several “supporting” classes, then several compositions can be used. We shall use a special notation for a *supporting* composition: a small box attached to the *weak* class with “PK” in it will serve as the anchor for the supporting composition. The implication is that the key attribute(s) for the *supporting* class at the other end of the composition is part of the key of the weak class, along with any of the attributes of the weak class that are marked “PK.” As with weak entity sets, there can be several supporting compositions and classes, and those supporting classes could themselves be weak, in which case the rule just described is applied recursively.

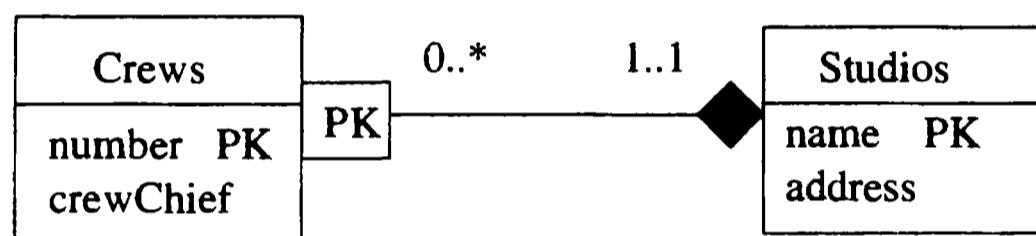


Figure 4.42: Weak class *Crews* supported by a composition and the class *Studios*

**Example 4.44:** Figure 4.42 shows the analog of the weak entity set *Crews* of Example 4.20. There is a composition from *Crews* to *Studios* anchored by a box labeled “PK” to indicate that this composition provides part of the key for *Crews*. □

We convert weak structures such as Fig. 4.42 to relations exactly as we did in Section 4.5.4. There is a relation for class *Studios* as usual. There is no relation for the composition, again as usual. The relation for class *Crews* includes not only its own attribute *number*, but the key for the class at the end of the composition, which is *Studios*.

**Example 4.45:** The relations for Example 4.44 are thus:

```

Studios(name, address)
Crews(number, crewChief, studioName)
  
```

As before, we renamed the attribute *name* of *Studios* in the *Crews* relation, for clarity. □

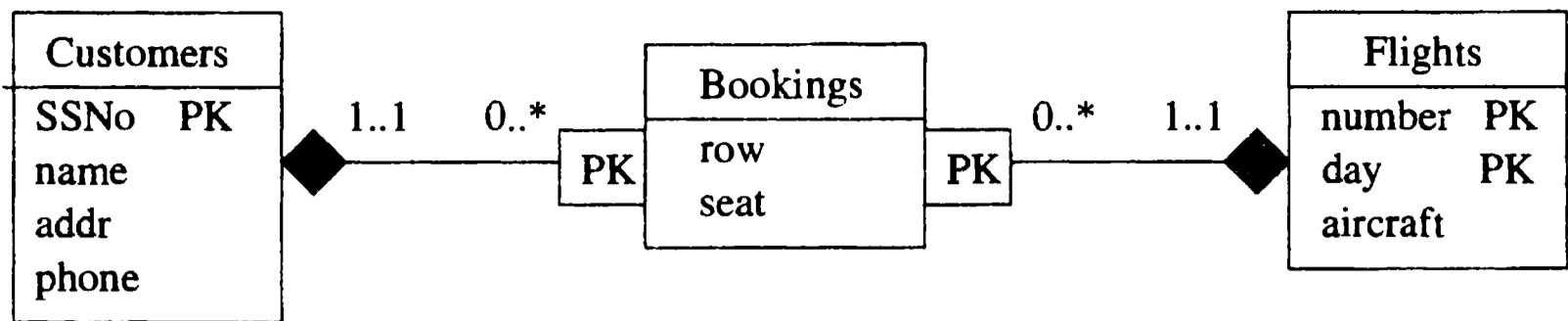


Figure 4.43: A UML diagram analogous to the E/R diagram of Fig. 4.29

#### 4.8.5 Exercises for Section 4.8

**Exercise 4.8.1:** Convert the UML diagram of Fig. 4.43 to relations.

**Exercise 4.8.2:** Convert the following UML diagrams to relations:

- a) Figure 4.37.
- b) Figure 4.40.
- c) Your solution to Exercise 4.7.1.
- d) Your solution to Exercise 4.7.3.
- e) Your solution to Exercise 4.7.4.
- f) Your solution to Exercise 4.7.6.

! **Exercise 4.8.3:** How many relations do we create, using the object-oriented approach, if we have a three-level hierarchy with three subclasses of each class at the first and second levels, and that hierarchy is:

- a) Disjoint and complete at each level.
- b) Disjoint but not complete at each level.
- c) Neither disjoint nor complete.

## 4.9 Object Definition Language

*ODL (Object Definition Language)* is a text-based language for specifying the structure of databases in object-oriented terms. Like UML, the class is the central concept in ODL. Classes in ODL have a name, attributes, and methods, just as UML classes do. Relationships, which are analogous to UML's associations, are not an independent concept in ODL, but are embedded within classes as an additional family of properties.

### 4.9.1 Class Declarations

A declaration of a class in ODL, in its simplest form, is:

```
class <name> {
    <list of properties>
};
```

That is, the keyword `class` is followed by the name of the class and a bracketed list of properties. A property can be an attribute, a relationship, or a method.

### 4.9.2 Attributes in ODL

The simplest kind of property is the *attribute*. In ODL, attributes need not be of simple types, such as integers and strings. ODL has a type system, described in Section 4.9.6, that allows us to form structured types and collection types (e.g., sets). For example, an attribute `address` might have a structured type with fields for the street, city, and zip code. An attribute `phones` might have a set of strings as its type, and even more complex types are possible. An attribute is represented in the declaration for its class by the keyword `attribute`, the type of the attribute, and the name of the attribute.

```
1)  class Movie {
2)      attribute string title;
3)      attribute integer year;
4)      attribute integer length;
5)      attribute enum Genres
            {drama, comedy, sciFi, teen} genre;
};
```

Figure 4.44: An ODL declaration of the class `Movie`

**Example 4.46:** In Fig. 4.44 is an ODL declaration of the class of movies. It is not a complete declaration; we shall add more to it later. Line (1) declares `Movie` to be a class. Following line (1) are the declarations of four attributes that all `Movie` objects will have.

Lines (2), (3), and (4) declare three attributes, `title`, `year`, and `length`. The first of these is of character-string type, and the other two are integers. Line (5) declares attribute `genre` to be of enumerated type. The name of the enumeration (list of symbolic constants) is `Genres`, and the four values the attribute `genre` is allowed to take are `drama`, `comedy`, `sciFi`, and `teen`. An enumeration must have a name, which can be used to refer to the same type anywhere. □

## Why Name Enumerations and Structures?

The enumeration-name `Genres` in Fig. 4.44 appears to play no role. However, by giving this set of symbolic constants a name, we can refer to it elsewhere, including in the declaration of other classes. In some other class, the *scoped name* `Movie::Genres` can be used to refer to the definition of the enumerated type of this name within the class `Movie`.

**Example 4.47:** In Example 4.46, all the attributes have primitive types. Here is an example with a complex type. We can define the class `Star` by

```

1) class Star {
2)     attribute string name;
3)     attribute Struct Addr
               {string street, string city} address;
};
```

Line (2) specifies an attribute `name` (of the star) that is a string. Line (3) specifies another attribute `address`. This attribute has a type that is a *record structure*. The name of this structure is `Addr`, and the type consists of two fields: `street` and `city`. Both fields are strings. In general, one can define record structure types in ODL by the keyword `Struct` and curly braces around the list of field names and their types. Like enumerations, structure types must have a name, which can be used elsewhere to refer to the same structure type.  
□

### 4.9.3 Relationships in ODL

An ODL relationship is declared inside a class declaration, by the keyword `relationship`, a type, and the name of the relationship. The type of a relationship describes what a single object of the class is connected to by the relationship. Typically, this type is either another class (if the relationship is many-one) or a collection type (if the relationship is one-many or many-many). We shall show complex types by example, until the full type system is described in Section 4.9.6.

**Example 4.48:** Suppose we want to add to the declaration of the `Movie` class from Example 4.46 a property that is a set of stars. More precisely, we want each `Movie` object to connect the set of `Star` objects that are its stars. The best way to represent this connection between the `Movie` and `Star` classes is with a *relationship*. We may represent this relationship by a line:

```
relationship Set<Star> stars;
```

in the declaration of class `Movie`. It says that in each object of class `Movie` there is a set of references to `Star` objects. The set of references is called `stars`. □

#### 4.9.4 Inverse Relationships

Just as we might like to access the stars of a given movie, we might like to know the movies in which a given star acted. To get this information into `Star` objects, we can add the line

```
relationship Set<Movie> starredIn;
```

to the declaration of class `Star` in Example 4.47. However, this line and a similar declaration for `Movie` omits a very important aspect of the relationship between movies and stars. We expect that if a star  $S$  is in the `stars` set for movie  $M$ , then movie  $M$  is in the `starredIn` set for star  $S$ . We indicate this connection between the relationships `stars` and `starredIn` by placing in each of their declarations the keyword `inverse` and the name of the other relationship. If the other relationship is in some other class, as it usually is, then we refer to that relationship by its scoped name — the name of its class, followed by a double colon (::) and the name of the relationship.

**Example 4.49:** To define the relationship `starredIn` of class `Star` to be the inverse of the relationship `stars` in class `Movie`, we revise the declarations of these classes, as shown in Fig. 4.45 (which also contains a definition of class `Studio` to be discussed later). Line (6) shows the declaration of relationship `stars` of movies, and says that its inverse is `Star::starredIn`. Since relationship `starredIn` is defined in another class, its scoped name must be used.

Similarly, relationship `starredIn` is declared in line (11). Its inverse is declared by that line to be `stars` of class `Movie`, as it must be, because inverses always are linked in pairs. □

As a general rule, if a relationship  $R$  for class  $C$  associates with object  $x$  of class  $C$  with objects  $y_1, y_2, \dots, y_n$  of class  $D$ , then the inverse relationship of  $R$  associates with each of the  $y_i$ 's the object  $x$  (perhaps along with other objects).

#### 4.9.5 Multiplicity of Relationships

Like the binary relationships of the E/R model, a pair of inverse relationships in ODL can be classified as either many-many, many-one in either direction, or one-one. The type declarations for the pair of relationships tells us which.

1. If we have a many-many relationship between classes  $C$  and  $D$ , then in class  $C$  the type of the relationship is `Set<D>`, and in class  $D$  the type is `Set<C>`.<sup>6</sup>

---

<sup>6</sup>Actually, the `Set` could be replaced by another “collection type,” such as `list` or `bag`, as discussed in Section 4.9.6. We shall assume all collections are sets in our exposition of relationships, however.

```

1) class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Genres
6)         {drama, comedy, sciFi, teen} genre;
7)     relationship Set<Star> stars
8)         inverse Star::starredIn;
9)     relationship Studio ownedBy
10)        inverse Studio::owns;
11)    };
12)
13) class Star {
14)     attribute string name;
15)     attribute Struct Addr
16)         {string street, string city} address;
17)     relationship Set<Movie> starredIn
18)         inverse Movie::stars;
19)    };
20)
21) class Studio {
22)     attribute string name;
23)     attribute Star::Addr address;
24)     relationship Set<Movie> owns
25)         inverse Movie::ownedBy;
26)    };

```

Figure 4.45: Some ODL classes and their relationships

2. If the relationship is many-one from  $C$  to  $D$ , then the type of the relationship in  $C$  is just  $D$ , while the type of the relationship in  $D$  is  $\text{Set}\langle C \rangle$ .
3. If the relationship is many-one from  $D$  to  $C$ , then the roles of  $C$  and  $D$  are reversed in (2) above.
4. If the relationship is one-one, then the type of the relationship in  $C$  is just  $D$ , and in  $D$  it is just  $C$ .

Note that, as in the E/R model, we allow a many-one or one-one relationship to include the case where for some objects the “one” is actually “none.” For instance, a many-one relationship from  $C$  to  $D$  might have a “null” value of the relationship in some of the  $C$  objects. Of course, since a  $D$  object could be associated with any set of  $C$  objects, it is also permissible for that set to be empty for some  $D$  objects.

**Example 4.50:** In Fig. 4.45 we have the declaration of three classes, `Movie`, `Star`, and `Studio`. The first two of these have already been introduced in Examples 4.46 and 4.47. We also discussed the relationship pair `stars` and `starredIn`. Since each of their types uses `Set`, we see that this pair represents a many-many relationship between `Star` and `Movie`.

`Studio` objects have attributes `name` and `address`; these appear in lines (13) and (14). We have used the same type for addresses of studios as we defined in class `Star` for addresses of stars.

In line (7) we see a relationship `ownedBy` from movies to studios, and the inverse of this relationship is `owns` on line (15). Since the type of `ownedBy` is `Studio`, while the type of `owns` is `Set<Movie>`, we see that this pair of inverse relationships is many-one from `Movie` to `Studio`.  $\square$

#### 4.9.6 Types in ODL

ODL offers the database designer a type system similar to that found in C or other conventional programming languages. A type system is built from a basis of types that are defined by themselves and certain recursive rules whereby complex types are built from simpler types. In ODL, the basis consists of:

1. *Primitive types*: integer, float, character, character string, boolean, and *enumerations*. The latter are lists of symbolic names, such as `drama` in line (5) of Fig. 4.45.
2. *Class names*, such as `Movie`, or `Star`, which represent types that are actually structures, with components for each of the attributes and relationships of that class.

These types are combined into structured types using the following *type constructors*:

1. *Set*. If  $T$  is any type, then `Set<T>` denotes the type whose values are finite sets of elements of type  $T$ . Examples using the set type-constructor occur in lines (6), (11), and (15) of Fig. 4.45.
2. *Bag*. If  $T$  is any type, then `Bag<T>` denotes the type whose values are finite bags or *multisets* of elements of type  $T$ .
3. *List*. If  $T$  is any type, then `List<T>` denotes the type whose values are finite lists of zero or more elements of type  $T$ .
4. *Array*. If  $T$  is a type and  $i$  is an integer, then `Array<T, i>` denotes the type whose elements are arrays of  $i$  elements of type  $T$ . For example, `Array<char, 10>` denotes character strings of length 10.
5. *Dictionary*. If  $T$  and  $S$  are types, then `Dictionary<T, S>` denotes a type whose values are finite sets of pairs. Each pair consists of a value of the *key type*  $T$  and a value of the *range type*  $S$ . The dictionary may not contain two pairs with the same key value.

## Sets, Bags, and Lists

To understand the distinction between sets, bags, and lists, remember that a set has unordered elements, and only one occurrence of each element. A bag allows more than one occurrence of an element, but the elements and their occurrences are unordered. A list allows more than one occurrence of an element, but the occurrences are ordered. Thus,  $\{1, 2, 1\}$  and  $\{2, 1, 1\}$  are the same bag, but  $(1, 2, 1)$  and  $(2, 1, 1)$  are not the same list.

6. *Structures.* If  $T_1, T_2, \dots, T_n$  are types, and  $F_1, F_2, \dots, F_n$  are names of fields, then

**Struct N { $T_1 F_1, T_2 F_2, \dots, T_n F_n$ }**

denotes the type named  $N$  whose elements are structures with  $n$  fields. The  $i$ th field is named  $F_i$  and has type  $T_i$ . For example, line (10) of Fig. 4.45 showed a structure type named **Addr**, with two fields. Both fields are of type **string** and have names **street** and **city**, respectively.

The first five types — set, bag, list, array, and dictionary — are called *collection types*. There are different rules about which types may be associated with attributes and which with relationships.

- The type of a relationship is either a class type or a single use of a collection type constructor applied to a class type.
- The type of an attribute is built starting with a primitive type or types.<sup>7</sup> We may then apply the structure and collection type constructors as we wish, as many times as we wish.

**Example 4.51:** Some of the possible types of attributes are:

1. **integer**.
2. **Struct N {string field1, integer field2}.**
3. **List<real>**.
4. **Array<Struct N {string field1, integer field2}, 10>**.

<sup>7</sup>Class types may also be used, which makes the attribute behave like a “one-way” relationship. We shall not consider such attributes here.

Example (1) is a primitive type, (2) is a structure of primitive types, (3) a collection of a primitive type, and (4) a collection of structures built from primitive types.

Now, suppose the class names `Movie` and `Star` are available primitive types. Then we may construct relationship types such as `Movie` or `Bag<Star>`. However, the following are illegal as relationship types:

1. `Struct N {Movie field1, Star field2}`. Relationship types cannot involve structures.
2. `Set<integer>`. Relationship types cannot involve primitive types.
3. `Set<Array<Star, 10>>`. Relationship types cannot involve two applications of collection types.

□

#### 4.9.7 Subclasses in ODL

We can declare one class  $C$  to be a subclass of another class  $D$ . To do so, follow the name  $C$  in its declaration with the keyword `extends` and the name  $D$ . Then, class  $C$  inherits all the properties of  $D$ , and may have additional properties of its own.

**Example 4.52:** Recall Example 4.10, where we declared `cartoons` to be a subclass of `movies`, with the additional property of a relationship from a cartoon to a set of stars that are its “voices.” We can create a subclass `Cartoon` for `Movie` with the ODL declaration:

```
class Cartoon extends Movie {
    relationship Set<Star> voices;
};
```

Also in that example, we defined a class of murder mysteries with additional attribute `weapon`.

```
class MurderMystery extends Movie {
    attribute string weapon;
};
```

is a suitable declaration of this subclass. □

Sometimes, as in the case of a movie like “Roger Rabbit,” we need a class that is a subclass of two or more other classes at the same time. In ODL, we may follow the keyword `extends` by several classes, separated by colons.<sup>8</sup> Thus, we may declare a fourth class by:

---

<sup>8</sup>Technically, the second and subsequent names must be “interfaces,” rather than classes. Roughly, an *interface* in ODL is a class definition without an associated set of objects.

```
class CartoonMurderMystery
    extends MurderMystery : Cartoon;
```

Note that when there is multiple inheritance, there is the potential for a class to inherit two properties with the same name. The way such conflicts are resolved is implementation-dependent.

#### 4.9.8 Declaring Keys in ODL

The declaration of a key or keys for a class is optional. The reason is that ODL, being object-oriented, assumes that all objects have an object-identity, as discussed in connection with UML in Section 4.8.4.

In ODL we may declare one or more attributes to be a key for a class by using the keyword **key** or **keys** (it doesn't matter which) followed by the attribute or attributes forming keys. If there is more than one attribute in a key, the list of attributes must be surrounded by parentheses. The key declaration itself appears inside parentheses, following the name of the class itself in the first line of its declaration.

**Example 4.53:** To declare that the set of two attributes **title** and **year** form a key for class **Movie**, we could begin its declaration:

```
class Movie (key (title, year)) {
```

We could have used **keys** in place of **key**, even though only one key is declared.

It is possible that several sets of attributes are keys. If so, then following the word **key(s)** we may place several keys separated by commas. A key that consists of more than one attribute must have parentheses around the list of its attributes, so we can disambiguate a key of several attributes from several keys of one attribute each.

The ODL standard also allows properties other than attributes to appear in keys. There is no fundamental problem with a method or relationship being declared a key or part of a key, since keys are advisory statements that the DBMS can take advantage of or not, as it wishes. For instance, one could declare a method to be a key, meaning that on distinct objects of the class the method is guaranteed to return distinct values.

When we allow many-one relationships to appear in key declarations, we can get an effect similar to that of weak entity sets in the E/R model. We can declare that the object  $O_1$  referred to by an object  $O_2$  on the "many" side of the relationship, perhaps together with other properties of  $O_2$  that are included in the key, is unique for different objects  $O_2$ . However, we should remember that there is no requirement that classes have keys; we are never obliged to handle, in some special way, classes that lack attributes of their own to form a key, as we did for weak entity sets.

**Example 4.54:** Let us review the example of a weak entity set *Crews* in Fig. 4.20. Recall that we hypothesized that crews were identified by their number, and the studio for which they worked, although two studios might have crews with the same number. We might declare the class *Crew* as in Fig. 4.46. Note that we should modify the declaration of *Studio* to include the relationship *crewsOf* that is an inverse to the relationship *unitOf* in *Crew*; we omit this change.

```
class Crew (key (number, unitOf)) {
    attribute integer number;
    attribute string crewChief;
    relationship Studio unitOf
        inverse Studio::crewsOf;
};
```

Figure 4.46: A ODL declaration for crews

What this key declaration asserts is that there cannot be two crews that both have the same value for the *number* attribute and are related to the same studio by *unitOf*. Notice how this assertion resembles the implication of the E/R diagram in Fig. 4.20, which is that the number of a crew and the name of the related studio (i.e., the key for studios) uniquely determine a crew entity. □

#### 4.9.9 Exercises for Section 4.9

**Exercise 4.9.1:** In Exercise 4.1.1 was the informal description of a bank database. Render this design in ODL, including keys as appropriate.

**Exercise 4.9.2:** Modify your design of Exercise 4.9.1 in the ways enumerated in Exercise 4.1.2. Describe the changes; do not write a complete, new schema.

**Exercise 4.9.3:** Render the teams-players-fans database of Exercise 4.1.3 in ODL, including keys, as appropriate. Why does the complication about sets of team colors, which was mentioned in the original exercise, not present a problem in ODL?

! **Exercise 4.9.4:** Suppose we wish to keep a genealogy. We shall have one class, *Person*. The information we wish to record about persons includes their name (an attribute) and the following relationships: mother, father, and children. Give an ODL design for the *Person* class. Be sure to indicate the inverses of the relationships that, like *mother*, *father*, and *children*, are also relationships from *Person* to itself. Is the inverse of the *mother* relationship the *children* relationship? Why or why not? Describe each of the relationships and their inverses as sets of pairs.

**! Exercise 4.9.5:** Let us add to the design of Exercise 4.9.4 the attribute `education`. The value of this attribute is intended to be a collection of the degrees obtained by each person, including the name of the degree (e.g., B.S.), the school, and the date. This collection of structs could be a set, bag, list, or array. Describe the consequences of each of these four choices. What information could be gained or lost by making each choice? Is the information lost likely to be important in practice?

**Exercise 4.9.6:** In Exercise 4.4.4 we saw two examples of situations where weak entity sets were essential. Render these databases in ODL, including declarations for suitable keys.

**Exercise 4.9.7:** Give an ODL design for the registrar's database described in Exercise 4.1.9.

**!! Exercise 4.9.8:** Under what circumstances is a relationship its own inverse?

*Hint:* Think about the relationship as a set of pairs, as discussed in Section 4.9.4.

## 4.10 From ODL Designs to Relational Designs

ODL was actually intended as the data-definition part of a language standard for object-oriented DBMS's, analogous to the SQL CREATE TABLE statement. Indeed, there have been some attempts to implement such a system. However, it is also possible to see ODL as a text-based, high-level design notation, from which we eventually derive a relational database schema. Thus, in this section we shall consider how to convert ODL designs into relational designs.

Much of the process is similar to that we discussed for E/R diagrams in Section 4.5 and for UML in Section 4.8. Classes become relations, and relationships become relations that connect the key attributes of the classes involved in the relationship. Yet some new problems arise for ODL, including:

1. Entity sets must have keys, but there is no such guarantee for ODL classes.
2. While attributes in E/R, UML, and the relational model are of primitive type, there is no such constraint for ODL attributes.

### 4.10.1 From ODL Classes to Relations

As a starting point, let us assume that our goal is to have one relation for each class and for that relation to have one attribute for each property. We shall see many ways in which this approach must be modified, but for the moment, let us consider the simplest possible case, where we can indeed convert classes to relations and properties to attributes. The restrictions we assume are:

1. All properties of the class are attributes (not relationships or methods).

2. The types of the attributes are primitive (not structures or sets).

In this case, the ODL class looks almost like an entity set or a UML class. Although there might be no key for the ODL class, ODL assumes object-identity. We can create an artificial attribute to represent the object-identity and serve as a key for the relation; this issue was introduced for UML in Section 4.8.4.

**Example 4.55:** Figure 4.47 is an ODL description of movie executives. No key is listed, and we do not assume that `name` uniquely determines a movie executive (unlike stars, who will make sure their chosen name is unique).

```
class MovieExec {
    attribute string name;
    attribute string address;
    attribute integer netWorth;
};
```

Figure 4.47: The class `MovieExec`

We create a relation with the same name as the class. The relation has four attributes, one for each attribute of the class, and one for the object-identity:

```
MovieExecs(cert#, name, address, netWorth)
```

We use `cert#` as the key attribute, representing the object-identity. □

#### 4.10.2 Complex Attributes in Classes

Even when a class' properties are all attributes we may have some difficulty converting the class to a relation. The reason is that attributes in ODL can have complex types such as structures, sets, bags, or lists. On the other hand, a fundamental principle of the relational model is that a relation's attributes have a primitive type, such as numbers and strings. Thus, we must find some way to represent complex attribute types as relations.

Record structures whose fields are themselves primitive are the easiest to handle. We simply expand the structure definition, making one attribute of the relation for each field of the structure.

```
class Star (key name) {
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
};
```

Figure 4.48: Class with a structured attribute

**Example 4.56:** In Fig. 4.48 is a declaration for class **Star**, with only attributes as properties. The attribute **name** is of primitive type, but attribute **address** is a structure with two fields, **street** and **city**. We represent this class by the relation:

```
Star(name, street, city)
```

The key is **name**, and the attributes **street** and **city** represent the structure **address**.  $\square$

### 4.10.3 Representing Set-Valued Attributes

However, record structures are not the most complex kind of attribute that can appear in ODL class definitions. Values can also be built using type constructors **Set**, **Bag**, **List**, **Array**, and **Dictionary** from Section 4.9.6. Each presents its own problems when migrating to the relational model. We shall only discuss the **Set** constructor, which is the most common, in detail.

One approach to representing a set of values for an attribute *A* is to make one tuple for each value. That tuple includes the appropriate values for all the other attributes besides *A*. This approach works, although it is likely to produce unnormalized relations, as we shall see in the next example.

```
class Star (key name) {
    attribute string name;
    attribute Set<
        Struct Addr {string street, string city}
        > address;
    attribute Date birthdate;
};
```

Figure 4.49: Stars with a set of addresses and a birthdate

**Example 4.57:** Figure 4.49 shows a new definition of the class **Star**, in which we have allowed stars to have a set of addresses and also added a nonkey, primitive attribute **birthdate**. The **birthdate** attribute can be an attribute of the **Star** relation, whose schema now becomes:

```
Star(name, street, city, birthdate)
```

Unfortunately, this relation exhibits the sort of anomalies we saw in Section 3.3.1. If Carrie Fisher has two addresses, say a home and a beach house, then she is represented by two tuples in the relation **Star**. If Harrison Ford has an empty set of addresses, then he does not appear at all in **Star**. A typical set of tuples for **Star** is shown in Fig. 4.50.

<i>name</i>	<i>street</i>	<i>city</i>	<i>birthdate</i>
Carrie Fisher	123 Maple St.	Hollywood	9/9/99
Carrie Fisher	5 Locust Ln.	Malibu	9/9/99
Mark Hamill	456 Oak Rd.	Brentwood	8/8/88

Figure 4.50: Adding birthdates

Although **name** is a key for the class **Star**, our need to have several tuples for one star to represent all their addresses means that **name** is *not* a key for the relation **Star**. In fact, the key for that relation is {**name**, **street**, **city**}. Thus, the functional dependency

$$\text{name} \rightarrow \text{birthdate}$$

is a BCNF violation and the multivalued dependency

$$\text{name} \twoheadrightarrow \text{street city}$$

is a 4NF violation as well.  $\square$

There are several options regarding how to handle set-valued attributes that appear in a class declaration along with other attributes, set-valued or not. One approach is to separate out each set-valued attribute as if it were a many-many relationship between the objects of the class and the values that appear in the sets.

An alternative approach is to place all attributes, set-valued or not, in the schema for the relation, then use the normalization techniques of Sections 3.3 and 3.6 to eliminate the resulting BCNF and 4NF violations. Notice that any set-valued attribute in conjunction with any single-valued attribute leads to a BNCF violation, as in Example 4.57. Two set-valued attributes in the same class declaration will lead to a 4NF violation, even if there are no single-valued attributes.

#### 4.10.4 Representing Other Type Constructors

Besides record structures and sets, an ODL class definition could use **Bag**, **List**, **Array**, or **Dictionary** to construct values. To represent a bag (multiset), in which a single object can be a member of the bag *n* times, we cannot simply introduce into a relation *n* identical tuples.<sup>9</sup> Instead, we could add to the relation schema another attribute **count** representing the number of times that

---

<sup>9</sup>To be precise, we cannot introduce identical tuples into relations of the abstract relational model described in Section 2.2. However, SQL-based relational DBMS's *do* allow duplicate tuples; i.e., relations are bags rather than sets in SQL. See Sections 5.1 and 6.4. If queries are likely to ask for tuple counts, we advise using a scheme such as that described here, even if your DBMS allows duplicate tuples.

each element is a member of the bag. For instance, suppose that **address** in Fig. 4.49 were a bag instead of a set. We could say that 123 Maple St., Hollywood is Carrie Fisher's address twice and 5 Locust Ln., Malibu is her address 3 times (whatever that may mean) by

<i>name</i>	<i>street</i>	<i>city</i>	<i>count</i>
Carrie Fisher	123 Maple St.	Hollywood	2
Carrie Fisher	5 Locust Ln.	Malibu	3

A list of addresses could be represented by a new attribute **position**, indicating the position in the list. For instance, we could show Carrie Fisher's addresses as a list, with Hollywood first, by:

<i>name</i>	<i>street</i>	<i>city</i>	<i>position</i>
Carrie Fisher	123 Maple St.	Hollywood	1
Carrie Fisher	5 Locust Ln.	Malibu	2

A fixed-length array of addresses could be represented by attributes for each position in the array. For instance, if **address** were to be an array of two street-city structures, we could represent **Star** objects as:

<i>name</i>	<i>street1</i>	<i>city1</i>	<i>street2</i>	<i>city2</i>
Carrie Fisher	123 Maple St.	Hollywood	5 Locust Ln.	Malibu

Finally, a dictionary could be represented as a set, but with attributes for both the key-value and range-value components of the pairs that are members of the dictionary. For instance, suppose that instead of star's addresses, we really wanted to keep, for each star, a dictionary giving the mortgage holder for each of their homes. Then the dictionary would have address as the key value and bank name as the range value. A hypothetical rendering of the Carrie-Fisher object with a dictionary attribute is:

<i>name</i>	<i>street</i>	<i>city</i>	<i>mortgage-holder</i>
Carrie Fisher	123 Maple St.	Hollywood	Bank of Burbank
Carrie Fisher	5 Locust Ln.	Malibu	Torrance Trust

Of course attribute types in ODL may involve more than one type constructor. If a type is any collection type besides dictionary applied to a structure (e.g., a set of structs), then we may apply the techniques from Sections 4.10.3 or 4.10.4 as if the struct were an atomic value, and then replace the single attribute representing the atomic value by several attributes, one for each field of the struct. This strategy was used in the examples above, where the address is a struct. The case of a dictionary applied to structs is similar and left as an exercise.

There are many reasons to limit the complexity of attribute types to an optional struct followed by an optional collection type. We mentioned in Section 4.1.1 that some versions of the E/R model allow exactly this much generality in the types of attributes, although we restricted ourselves to attributes of

primitive type in the E/R model. We recommend that, if you are going to use an ODL design for the purpose of eventual translation to a relational database schema, you similarly limit yourself. We take up in the exercises some options for dealing with more complex types as attributes.

#### 4.10.5 Representing ODL Relationships

Usually, an ODL class definition will contain relationships to other ODL classes. As in the E/R model, we can create for each relationship a new relation that connects the keys of the two related classes. However, in ODL, relationships come in inverse pairs, and we must create only one relation for each pair.

When a relationship is many-one, we have an option to combine it with the relation that is constructed for the class on the “many” side. Doing so has the effect of combining two relations that have a common key, as we discussed in Section 4.5.3. It therefore does not cause a BCNF violation and is a legitimate and commonly followed option.

#### 4.10.6 Exercises for Section 4.10

**Exercise 4.10.1:** Convert your ODL designs from the following exercises to relational database schemas.

- a) Exercise 4.9.1.
- b) Exercise 4.9.2 (include all four of the modifications specified by that exercise).
- c) Exercise 4.9.3.
- d) Exercise 4.9.4.
- e) Exercise 4.9.5.

**! Exercise 4.10.2:** Consider an attribute of type `Dictionary` with key and range types both structs of primitive types. Show how to convert a class with an attribute of this type to a relation.

**Exercise 4.10.3:** We mentioned that when attributes are of a type more complex than a collection of structs, it becomes tricky to convert them to relations; in particular, it becomes necessary to create some intermediate concepts and relations for them. The following sequence of questions will examine increasingly more complex types and how to represent them as relations.

- a) A *card* can be represented as a struct with fields `rank` (2, 3, . . . , 10, Jack, Queen, King, and Ace) and `suit` (Clubs, Diamonds, Hearts, and Spades). Give a suitable definition of a structured type `Card`. This definition should be independent of any class declarations but available to them all.

- b) A *hand* is a set of cards. The number of cards may vary. Give a declaration of a class **Hand** whose objects are hands. That is, this class declaration has an attribute **theHand**, whose type is a hand.
- ! c) Convert your class declaration **Hand** from (b) to a relation schema.
- d) A *poker hand* is a set of five cards. Repeat (b) and (c) for poker hands.
- ! e) A *deal* is a set of pairs, each pair consisting of the name of a player and a hand for that player. Declare a class **Deal**, whose objects are deals. That is, this class declaration has an attribute **theDeal**, whose type is a deal.
- f) Repeat (e), but restrict hands of a deal to be hands of exactly five cards.
- g) Repeat (e), using a dictionary for a deal. You may assume the names of players in a deal are unique.
- !! h) Convert your class declaration from (e) to a relational database schema.
- ! i) Suppose we defined deals to be sets of sets of cards, with no player associated with each hand (set of cards). It is proposed that we represent such deals by a relation schema **Deals(dealID, card)**, meaning that the card was a member of one of the hands in the deal with the given ID. What, if anything, is wrong with this representation? How would you fix the problem?

**Exercise 4.10.4:** Suppose we have a class *C* defined by

```
class C (key a) {
    attribute string a;
    attribute T b;
};
```

where *T* is some type. Give the relation schema for the relation derived from *C* and indicate its key attributes if *T* is:

- a) **Set<Struct S {string f, string g}>**
- ! b) **Bag<Struct S {string f, string g}>**
- ! c) **List<Struct S {string f, string }>**
- ! d) **Dictionary<Struct K {string f, string g}, Struct R {string i, string j}>**

## 4.11 Summary of Chapter 4

- ◆ *The Entity-Relationship Model:* In the E/R model we describe entity sets, relationships among entity sets, and attributes of entity sets and relationships. Members of entity sets are called entities.
- ◆ *Entity-Relationship Diagrams:* We use rectangles, diamonds, and ovals to draw entity sets, relationships, and attributes, respectively.
- ◆ *Multiplicity of Relationships:* Binary relationships can be one-one, many-one, or many-many. In a one-one relationship, an entity of either set can be associated with at most one entity of the other set. In a many-one relationship, each entity of the “many” side is associated with at most one entity of the other side. Many-many relationships place no restriction.
- ◆ *Good Design:* Designing databases effectively requires that we represent the real world faithfully, that we select appropriate elements (e.g., relationships, attributes), and that we avoid redundancy — saying the same thing twice or saying something in an indirect or overly complex manner.
- ◆ *Subclasses:* The E/R model uses a special relationship *isa* to represent the fact that one entity set is a special case of another. Entity sets may be connected in a hierarchy with each child node a special case of its parent. Entities may have components belonging to any subtree of the hierarchy, as long as the subtree includes the root.
- ◆ *Weak Entity Sets:* These require attributes of some supporting entity set(s) to identify their own entities. A special notation involving diamonds and rectangles with double borders is used to distinguish weak entity sets.
- ◆ *Converting Entity Sets to Relations:* The relation for an entity set has one attribute for each attribute of the entity set. An exception is a weak entity set  $E$ , whose relation must also have attributes for the key attributes of its supporting entity sets.
- ◆ *Converting Relationships to Relations:* The relation for an E/R relationship has attributes corresponding to the key attributes of each entity set that participates in the relationship. However, if a relationship is a supporting relationship for some weak entity set, it is not necessary to produce a relation for that relationship.
- ◆ *Converting Isa Hierarchies to Relations:* One approach is to create a relation for each entity set with the key attributes of the hierarchy’s root plus the attributes of the entity set itself. A second approach is to create a relation for each possible subset of the entity sets in the hierarchy, and create for each entity one tuple; that tuple is in the relation for exactly the set of entity sets to which the entity belongs. A third approach is to create only one relation and to use null values for those attributes that do not apply to the entity represented by a given tuple.

- ◆ *Unified Modeling Language:* In UML, we describe classes and associations between classes. Classes are analogous to E/R entity sets, and associations are like binary E/R relationships. Special kinds of many-one associations, called aggregations and compositions, are used and have implications as to how they are translated to relations.
- ◆ *UML Subclass Hierarchies:* UML permits classes to have subclasses, with inheritance from the superclass. The subclasses of a class can be complete or partial, and they can be disjoint or overlapping.
- ◆ *Converting UML Diagrams to Relations:* The methods are similar to those used for the E/R model. Classes become relations and associations become relations connecting the keys of the associated classes. Aggregations and compositions are combined with the relation constructed from the class at the “many” end.
- ◆ *Object Definition Language:* This language is a notation for formally describing the schemas of databases in an object-oriented style. One defines classes, which may have three kinds of properties: attributes, methods, and relationships.
- ◆ *ODL Relationships:* A relationship in ODL must be binary. It is represented, in the two classes it connects, by names that are declared to be inverses of one another. Relationships can be many-many, many-one, or one-one, depending on whether the types of the pair are declared to be a single object or a set of objects.
- ◆ *The ODL Type System:* ODL allows types to be constructed, beginning with class names and atomic types such as integer, by applying any of the following type constructors: structure formation, set-of, bag-of, list-of, array-of, and dictionary-of.
- ◆ *Keys in ODL:* Keys are optional in ODL. We can declare one or more keys, but because objects have an object-ID that is not one of its properties, a system implementing ODL can tell the difference between objects, even if they have identical values for all properties.
- ◆ *Converting ODL Classes to Relations:* The method is the same as for E/R or UML, except if the class has attributes of complex type. If that happens the resulting relation may be unnormalized and will have to be decomposed. It may also be necessary to create a new attribute to represent the object-identity of objects and serve as a key.
- ◆ *Converting ODL Relationships to Relations:* The method is the same as for E/R relationships, except that we must first pair ODL relationships and their inverses, and create only one relation for the pair.

## 4.12 References for Chapter 4

The original paper on the Entity-Relationship model is [5]. Two books on the subject of E/R design are [2] and [7].

The manual defining ODL is [4]. One can also find more about the history of object-oriented database systems from [1], [3], and [6].

1. F. Bancilhon, C. Delobel, and P. Kanellakis, *Building an Object-Oriented Database System*, Morgan-Kaufmann, San Francisco, 1992.
2. Carlo Batini, S. Ceri, S. B. Navathe, and Carol Batini, *Conceptual Database Design: an Entity/Relationship Approach*, Addison-Wesley, Boston MA, 1991.
3. R. G. G. Cattell, *Object Data Management*, Addison-Wesley, Reading, MA, 1994.
4. R. G. G. Cattell (ed.), *The Object Database Standard: ODMG-99*, Morgan-Kaufmann, San Francisco, 1999.
5. P. P. Chen, “The entity-relationship model: toward a unified view of data,” *ACM Trans. on Database Systems* 1:1, pp. 9–36, 1976.
6. W. Kim (ed.), *Modern Database Systems: The Object Model, Interoperability, and Beyond*, ACM Press, New York, 1994.
7. B. Thalheim, “Fundamentals of Entity-Relationship Modeling,” Springer-Verlag, Berlin, 2000.

# **Part II**

# **Relational Database Programming**



# Chapter 5

# Algebraic and Logical Query Languages

We now switch our attention from modeling to programming for relational databases. We start in this discussion with two abstract programming languages, one algebraic and the other logic-based. The algebraic programming language, relational algebra, was introduced in Section 2.4, to let us see what operations in the relational model look like. However, there is more to the algebra. In this chapter, we extend the set-based algebra of Section 2.4 to bags, which better reflect the way the relational model is implemented in practice. We also extend the algebra so it can handle several more operations than were described previously; for example, we need to do aggregations (e.g., averages) of columns of a relation.

We close the chapter with another form of query language, based on logic. This language, called “Datalog,” allows us to express queries by describing the desired results, rather than by giving an algorithm to compute the results, as relational algebra requires.

## 5.1 Relational Operations on Bags

In this section, we shall consider relations that are bags (multisets) rather than sets. That is, we shall allow the same tuple to appear more than once in a relation. When relations are bags, there are changes that need to be made to the definition of some relational operations, as we shall see. First, let us look at a simple example of a relation that is a bag but not a set.

**Example 5.1:** The relation in Fig. 5.1 is a bag of tuples. In it, the tuple  $(1, 2)$  appears three times and the tuple  $(3, 4)$  appears once. If Fig. 5.1 were a set-valued relation, we would have to eliminate two occurrences of the tuple  $(1, 2)$ . In a bag-valued relation, we *do* allow multiple occurrences of the same tuple, but like sets, the order of tuples does not matter.  $\square$

<i>A</i>	<i>B</i>
1	2
3	4
1	2
1	2

Figure 5.1: A bag

### 5.1.1 Why Bags?

As we mentioned, commercial DBMS's implement relations that are bags, rather than sets. An important motivation for relations as bags is that some relational operations are considerably more efficient if we use the bag model. For example:

1. To take the union of two relations as bags, we simply copy one relation and add to the copy all the tuples of the other relation. There is no need to eliminate duplicate copies of a tuple that happens to be in both relations.
2. When we project relation as sets, we need to compare each projected tuple with all the other projected tuples, to make sure that each projection appears only once. However, if we can accept a bag as the result, then we simply project each tuple and add it to the result; no comparison with other projected tuples is necessary.

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

Figure 5.2: Bag for Example 5.2

**Example 5.2:** The bag of Fig. 5.1 could be the result of projecting the relation shown in Fig. 5.2 onto attributes *A* and *B*, provided we allow the result to be a bag and do not eliminate the duplicate occurrences of (1, 2). Had we used the ordinary projection operator of relational algebra, and therefore eliminated duplicates, the result would be only:

<i>A</i>	<i>B</i>
1	2
3	4

Note that the bag result, although larger, can be computed more quickly, since there is no need to compare each tuple  $(1, 2)$  or  $(3, 4)$  with previously generated tuples.  $\square$

Another motivation for relations as bags is that there are some situations where the expected answer can only be obtained if we use bags, at least temporarily. Here is an example.

**Example 5.3:** Suppose we want to take the average of the  $A$ -components of a set-valued relation such as Fig. 5.2. We could not use the set model to think of the relation projected onto attribute  $A$ . As a set, the average value of  $A$  is 2, because there are only two values of  $A$  — 1 and 3 — in Fig. 5.2, and their average is 2. However, if we treat the  $A$ -column in Fig. 5.2 as a bag  $\{1, 3, 1, 1\}$ , we get the correct average of  $A$ , which is 1.5, among the four tuples of Fig. 5.2.  $\square$

### 5.1.2 Union, Intersection, and Difference of Bags

These three operations have new definitions for bags. Suppose that  $R$  and  $S$  are bags, and that tuple  $t$  appears  $n$  times in  $R$  and  $m$  times in  $S$ . Note that either  $n$  or  $m$  (or both) can be 0. Then:

- In the bag union  $R \cup S$ , tuple  $t$  appears  $n + m$  times.
- In the bag intersection  $R \cap S$ , tuple  $t$  appears  $\min(n, m)$  times.
- In the bag difference  $R - S$ , tuple  $t$  appears  $\max(0, n - m)$  times. That is, if tuple  $t$  appears in  $R$  more times than it appears in  $S$ , then  $t$  appears in  $R - S$  the number of times it appears in  $R$ , minus the number of times it appears in  $S$ . However, if  $t$  appears at least as many times in  $S$  as it appears in  $R$ , then  $t$  does not appear at all in  $R - S$ . Intuitively, occurrences of  $t$  in  $S$  each “cancel” one occurrence in  $R$ .

**Example 5.4:** Let  $R$  be the relation of Fig. 5.1, that is, a bag in which tuple  $(1, 2)$  appears three times and  $(3, 4)$  appears once. Let  $S$  be the bag

$A$	$B$
1	2
3	4
3	4
5	6

Then the bag union  $R \cup S$  is the bag in which  $(1, 2)$  appears four times (three times for its occurrences in  $R$  and once for its occurrence in  $S$ );  $(3, 4)$  appears three times, and  $(5, 6)$  appears once.

The bag intersection  $R \cap S$  is the bag

<i>A</i>	<i>B</i>
1	2
3	4

with one occurrence each of  $(1, 2)$  and  $(3, 4)$ . That is,  $(1, 2)$  appears three times in  $R$  and once in  $S$ , and  $\min(3, 1) = 1$ , so  $(1, 2)$  appears once in  $R \cap S$ . Similarly,  $(3, 4)$  appears  $\min(1, 2) = 1$  time in  $R \cap S$ . Tuple  $(5, 6)$ , which appears once in  $S$  but zero times in  $R$  appears  $\min(0, 1) = 0$  times in  $R \cap S$ . In this case, the result happens to be a set, but any set is also a bag.

The bag difference  $R - S$  is the bag

<i>A</i>	<i>B</i>
1	2
1	2

To see why, notice that  $(1, 2)$  appears three times in  $R$  and once in  $S$ , so in  $R - S$  it appears  $\max(0, 3 - 1) = 2$  times. Tuple  $(3, 4)$  appears once in  $R$  and twice in  $S$ , so in  $R - S$  it appears  $\max(0, 1 - 2) = 0$  times. No other tuple appears in  $R$ , so there can be no other tuples in  $R - S$ .

As another example, the bag difference  $S - R$  is the bag

<i>A</i>	<i>B</i>
3	4
5	6

Tuple  $(3, 4)$  appears once because that is the number of times it appears in  $S$  minus the number of times it appears in  $R$ . Tuple  $(5, 6)$  appears once in  $S - R$  for the same reason.  $\square$

### 5.1.3 Projection of Bags

We have already illustrated the projection of bags. As we saw in Example 5.2, each tuple is processed independently during the projection. If  $R$  is the bag of Fig. 5.2 and we compute the bag-projection  $\pi_{A,B}(R)$ , then we get the bag of Fig. 5.1.

If the elimination of one or more attributes during the projection causes the same tuple to be created from several tuples, these duplicate tuples are not eliminated from the result of a bag-projection. Thus, the three tuples  $(1, 2, 5)$ ,  $(1, 2, 7)$ , and  $(1, 2, 8)$  of the relation  $R$  from Fig. 5.2 each gave rise to the same tuple  $(1, 2)$  after projection onto attributes  $A$  and  $B$ . In the bag result, there are three occurrences of tuple  $(1, 2)$ , while in the set-projection, this tuple appears only once.

## Bag Operations on Sets

Imagine we have two sets  $R$  and  $S$ . Every set may be thought of as a bag; the bag just happens to have at most one occurrence of any tuple. Suppose we intersect  $R \cap S$ , but we think of  $R$  and  $S$  as bags and use the bag intersection rule. Then we get the same result as we would get if we thought of  $R$  and  $S$  as sets. That is, thinking of  $R$  and  $S$  as bags, a tuple  $t$  is in  $R \cap S$  the minimum of the number of times it is in  $R$  and  $S$ . Since  $R$  and  $S$  are sets,  $t$  can be in each only 0 or 1 times. Whether we use the bag or set intersection rules, we find that  $t$  can appear at most once in  $R \cap S$ , and it appears once exactly when it is in both  $R$  and  $S$ . Similarly, if we use the bag difference rule to compute  $R - S$  or  $S - R$  we get exactly the same result as if we used the set rule.

However, union behaves differently, depending on whether we think of  $R$  and  $S$  as sets or bags. If we use the bag rule to compute  $R \cup S$ , then the result may not be a set, even if  $R$  and  $S$  are sets. In particular, if tuple  $t$  appears in both  $R$  and  $S$ , then  $t$  appears twice in  $R \cup S$  if we use the bag rule for union. But if we use the set rule then  $t$  appears only once in  $R \cup S$ .

### 5.1.4 Selection on Bags

To apply a selection to a bag, we apply the selection condition to each tuple independently. As always with bags, we do not eliminate duplicate tuples in the result.

**Example 5.5:** If  $R$  is the bag

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

then the result of the bag-selection  $\sigma_{C \geq 6}(R)$  is

A	B	C
3	4	6
1	2	7
1	2	7

That is, all but the first tuple meets the selection condition. The last two tuples, which are duplicates in  $R$ , are each included in the result.  $\square$

## Algebraic Laws for Bags

An algebraic law is an equivalence between two expressions of relational algebra whose arguments are variables standing for relations. The equivalence asserts that no matter what relations we substitute for these variables, the two expressions define the same relation. An example of a well-known law is the commutative law for union:  $R \cup S = S \cup R$ . This law happens to hold whether we regard relation-variables  $R$  and  $S$  as standing for sets or bags. However, there are a number of other laws that hold when relational algebra is applied to sets but that do not hold when relations are interpreted as bags. A simple example of such a law is the distributive law of set difference over union,  $(R \cup S) - T = (R - T) \cup (S - T)$ . This law holds for sets but not for bags. To see why it fails for bags, suppose  $R$ ,  $S$ , and  $T$  each have one copy of tuple  $t$ . Then the expression on the left has one  $t$ , while the expression on the right has none. As sets, neither would have  $t$ . Some exploration of algebraic laws for bags appears in Exercises 5.1.4 and 5.1.5.

### 5.1.5 Product of Bags

The rule for the Cartesian product of bags is the expected one. Each tuple of one relation is paired with each tuple of the other, regardless of whether it is a duplicate or not. As a result, if a tuple  $r$  appears in a relation  $R$   $m$  times, and tuple  $s$  appears  $n$  times in relation  $S$ , then in the product  $R \times S$ , the tuple  $rs$  will appear  $mn$  times.

**Example 5.6:** Let  $R$  and  $S$  be the bags shown in Fig. 5.3. Then the product  $R \times S$  consists of six tuples, as shown in Fig. 5.3(c). Note that the usual convention regarding attribute names that we developed for set-relations applies equally well to bags. Thus, the attribute  $B$ , which belongs to both relations  $R$  and  $S$ , appears twice in the product, each time prefixed by one of the relation names.  $\square$

### 5.1.6 Joins of Bags

Joining bags presents no surprises. We compare each tuple of one relation with each tuple of the other, decide whether or not this pair of tuples joins successfully, and if so we put the resulting tuple in the answer. When constructing the answer, we do not eliminate duplicate tuples.

**Example 5.7:** The natural join  $R \bowtie S$  of the relations  $R$  and  $S$  seen in Fig. 5.3 is

<i>A</i>	<i>B</i>
1	2
1	2

(a) The relation *R*

<i>B</i>	<i>C</i>
2	3
4	5
4	5

(b) The relation *S*

<i>A</i>	<i>R.B</i>	<i>S.B</i>	<i>C</i>
1	2	2	3
1	2	2	3
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

(c) The product  $R \times S$ 

Figure 5.3: Computing the product of bags

<i>A</i>	<i>B</i>	<i>C</i>
1	2	3
1	2	3

That is, tuple  $(1, 2)$  of *R* joins with  $(2, 3)$  of *S*. Since there are two copies of  $(1, 2)$  in *R* and one copy of  $(2, 3)$  in *S*, there are two pairs of tuples that join to give the tuple  $(1, 2, 3)$ . No other tuples from *R* and *S* join successfully.

As another example on the same relations *R* and *S*, the theta-join

$$R \bowtie_{R.B < S.B} S$$

produces the bag

<i>A</i>	<i>R.B</i>	<i>S.B</i>	<i>C</i>
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

The computation of the join is as follows. Tuple  $(1, 2)$  from  $R$  and  $(4, 5)$  from  $S$  meet the join condition. Since each appears twice in its relation, the number of times the joined tuple appears in the result is  $2 \times 2$  or 4. The other possible join of tuples —  $(1, 2)$  from  $R$  with  $(2, 3)$  from  $S$  — fails to meet the join condition, so this combination does not appear in the result.  $\square$

### 5.1.7 Exercises for Section 5.1

**Exercise 5.1.1:** Let  $PC$  be the relation of Fig. 2.21(a), and suppose we compute the projection  $\pi_{\text{speed}}(PC)$ . What is the value of this expression as a set? As a bag? What is the average value of tuples in this projection, when treated as a set? As a bag?

**Exercise 5.1.2:** Repeat Exercise 5.1.1 for the projection  $\pi_{hd}(PC)$ .

**Exercise 5.1.3:** This exercise refers to the “battleship” relations of Exercise 2.4.3.

- a) The expression  $\pi_{\text{bore}}(\text{Classes})$  yields a single-column relation with the bores of the various classes. For the data of Exercise 2.4.3, what is this relation as a set? As a bag?
- ! b) Write an expression of relational algebra to give the bores of the ships (not the classes). Your expression must make sense for bags; that is, the number of times a value  $b$  appears must be the number of ships that have bore  $b$ .
- ! **Exercise 5.1.4:** Certain algebraic laws for relations as sets also hold for relations as bags. Explain why each of the laws below hold for bags as well as sets.

- a) The associative law for union:  $(R \cup S) \cup T = R \cup (S \cup T)$ .
- b) The associative law for intersection:  $(R \cap S) \cap T = R \cap (S \cap T)$ .
- c) The associative law for natural join:  $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$ .
- d) The commutative law for union:  $(R \cup S) = (S \cup R)$ .
- e) The commutative law for intersection:  $(R \cap S) = (S \cap R)$ .
- f) The commutative law for natural join:  $(R \bowtie S) = (S \bowtie R)$ .
- g)  $\pi_L(R \cup S) = \pi_L(R) \cup \pi_L(S)$ . Here,  $L$  is an arbitrary list of attributes.
- h) The distributive law of union over intersection:

$$R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$$

- i)  $\sigma_{C \text{ AND } D}(R) = \sigma_C(R) \cap \sigma_D(R)$ . Here,  $C$  and  $D$  are arbitrary conditions about the tuples of  $R$ .

**!! Exercise 5.1.5:** The following algebraic laws hold for sets but not for bags. Explain why they hold for sets and give counterexamples to show that they do not hold for bags.

a)  $(R \cap S) - T = R \cap (S - T)$ .

b) The distributive law of intersection over union:

$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$$

- c)  $\sigma_{C \text{ OR } D}(R) = \sigma_C(R) \cup \sigma_D(R)$ . Here,  $C$  and  $D$  are arbitrary conditions about the tuples of  $R$ .

## 5.2 Extended Operators of Relational Algebra

Section 2.4 presented the classical relational algebra, and Section 5.1 introduced the modifications necessary to treat relations as bags of tuples rather than sets. The ideas of these two sections serve as a foundation for most of modern query languages. However, languages such as SQL have several other operations that have proved quite important in applications. Thus, a full treatment of relational operations must include a number of other operators, which we introduce in this section. The additions:

1. The *duplicate-elimination operator*  $\delta$  turns a bag into a set by eliminating all but one copy of each tuple.
2. *Aggregation operators*, such as sums or averages, are not operations of relational algebra, but are used by the grouping operator (described next). Aggregation operators apply to attributes (columns) of a relation; e.g., the sum of a column produces the one number that is the sum of all the values in that column.
3. *Grouping* of tuples according to their value in one or more attributes has the effect of partitioning the tuples of a relation into “groups.” Aggregation can then be applied to columns within each group, giving us the ability to express a number of queries that are impossible to express in the classical relational algebra. The *grouping operator*  $\gamma$  is an operator that combines the effect of grouping and aggregation.
4. *Extended projection* gives additional power to the operator  $\pi$ . In addition to projecting out some columns, in its generalized form  $\pi$  can perform computations involving the columns of its argument relation to produce new columns.

5. The *sorting operator*  $\tau$  turns a relation into a list of tuples, sorted according to one or more attributes. This operator should be used judiciously, because some relational-algebra operators do not make sense on lists. We can, however, apply selections or projections to lists and expect the order of elements on the list to be preserved in the output.
6. The *outerjoin* operator is a variant of the join that avoids losing dangling tuples. In the result of the outerjoin, dangling tuples are “padded” with the null value, so the dangling tuples can be represented in the output.

### 5.2.1 Duplicate Elimination

Sometimes, we need an operator that converts a bag to a set. For that purpose, we use  $\delta(R)$  to return the set consisting of one copy of every tuple that appears one or more times in relation  $R$ .

**Example 5.8:** If  $R$  is the relation

A	B
1	2
3	4
1	2
1	2

from Fig. 5.1, then  $\delta(R)$  is

A	B
1	2
3	4

Note that the tuple  $(1, 2)$ , which appeared three times in  $R$ , appears only once in  $\delta(R)$ .  $\square$

### 5.2.2 Aggregation Operators

There are several operators that apply to sets or bags of numbers or strings. These operators are used to summarize or “aggregate” the values in one column of a relation, and thus are referred to as *aggregation* operators. The standard operators of this type are:

1. **SUM** produces the sum of a column with numerical values.
2. **AVG** produces the average of a column with numerical values.
3. **MIN** and **MAX**, applied to a column with numerical values, produces the smallest or largest value, respectively. When applied to a column with character-string values, they produce the lexicographically (alphabetically) first or last value, respectively.

4. COUNT produces the number of (not necessarily distinct) values in a column. Equivalently, COUNT applied to any attribute of a relation produces the number of tuples of that relation, including duplicates.

**Example 5.9:** Consider the relation

A	B
1	2
3	4
1	2
1	2

Some examples of aggregations on the attributes of this relation are:

1.  $\text{SUM}(B) = 2 + 4 + 2 + 2 = 10$ .
2.  $\text{AVG}(A) = (1 + 3 + 1 + 1)/4 = 1.5$ .
3.  $\text{MIN}(A) = 1$ .
4.  $\text{MAX}(B) = 4$ .
5.  $\text{COUNT}(A) = 4$ .

□

### 5.2.3 Grouping

Often we do not want simply the average or some other aggregation of an entire column. Rather, we need to consider the tuples of a relation in groups, corresponding to the value of one or more other columns, and we aggregate only within each group. As an example, suppose we wanted to compute the total number of minutes of movies produced by each studio, i.e., a relation such as:

studioName	sumOfLengths
Disney	12345
MGM	54321
...	...

Starting with the relation

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

from our example database schema of Section 2.2.8, we must group the tuples according to their value for attribute `studioName`. We must then sum the `length` column within each group. That is, we imagine that the tuples of `Movies` are grouped as suggested in Fig. 5.4, and we apply the aggregation `SUM(length)` to each group independently.

	<i>studioName</i>
	Disney
	Disney
	Disney
	MGM
	MGM
	○
	○
	○

Figure 5.4: A relation with imaginary division into groups

### 5.2.4 The Grouping Operator

We shall now introduce an operator that allows us to group a relation and/or aggregate some columns. If there is grouping, then the aggregation is within groups.

The subscript used with the  $\gamma$  operator is a list  $L$  of elements, each of which is either:

- a) An attribute of the relation  $R$  to which the  $\gamma$  is applied; this attribute is one of the attributes by which  $R$  will be grouped. This element is said to be a *grouping attribute*.
- b) An aggregation operator applied to an attribute of the relation. To provide a name for the attribute corresponding to this aggregation in the result, an arrow and new name are appended to the aggregation. The underlying attribute is said to be an *aggregated attribute*.

The relation returned by the expression  $\gamma_L(R)$  is constructed as follows:

1. Partition the tuples of  $R$  into *groups*. Each group consists of all tuples having one particular assignment of values to the grouping attributes in the list  $L$ . If there are no grouping attributes, the entire relation  $R$  is one group.
2. For each group, produce one tuple consisting of:
  - i. The grouping attributes' values for that group and
  - ii. The aggregations, over all tuples of that group, for the aggregated attributes on list  $L$ .

**Example 5.10:** Suppose we have the relation

`StarsIn(title, year, starName)`

### $\delta$ is a Special Case of $\gamma$

Technically, the  $\delta$  operator is redundant. If  $R(A_1, A_2, \dots, A_n)$  is a relation, then  $\delta(R)$  is equivalent to  $\gamma_{A_1, A_2, \dots, A_n}(R)$ . That is, to eliminate duplicates, we group on all the attributes of the relation and do no aggregation. Then each group corresponds to a tuple that is found one or more times in  $R$ . Since the result of  $\gamma$  contains exactly one tuple from each group, the effect of this “grouping” is to eliminate duplicates. However, because  $\delta$  is such a common and important operator, we shall continue to consider it separately when we study algebraic laws and algorithms for implementing the operators.

One can also see  $\gamma$  as an extension of the projection operator on sets. That is,  $\gamma_{A_1, A_2, \dots, A_n}(R)$  is also the same as  $\pi_{A_1, A_2, \dots, A_n}(R)$ , if  $R$  is a set. However, if  $R$  is a bag, then  $\gamma$  eliminates duplicates while  $\pi$  does not.

and we wish to find, for each star who has appeared in at least three movies, the earliest year in which they appeared. The first step is to group, using **starName** as a grouping attribute. We clearly must compute for each group the **MIN(year)** aggregate. However, in order to decide which groups satisfy the condition that the star appears in at least three movies, we must also compute the **COUNT(title)** aggregate for each group.

We begin with the grouping expression

$$\gamma_{\text{starName}, \text{MIN}(\text{year}) \rightarrow \text{minYear}, \text{COUNT}(\text{title}) \rightarrow \text{ctTitle}}(\text{StarsIn})$$

The first two columns of the result of this expression are needed for the query result. The third column is an auxiliary attribute, which we have named **ctTitle**; it is needed to determine whether a star has appeared in at least three movies. That is, we continue the algebraic expression for the query by selecting for **ctTitle**  $\geq 3$  and then projecting onto the first two columns. An expression tree for the query is shown in Fig. 5.5.  $\square$

### 5.2.5 Extending the Projection Operator

Let us reconsider the projection operator  $\pi_L(R)$  introduced in Section 2.4.5. In the classical relational algebra,  $L$  is a list of (some of the) attributes of  $R$ . We extend the projection operator to allow it to compute with components of tuples as well as choose components. In *extended projection*, also denoted  $\pi_L(R)$ , projection lists can have the following kinds of elements:

1. A single attribute of  $R$ .

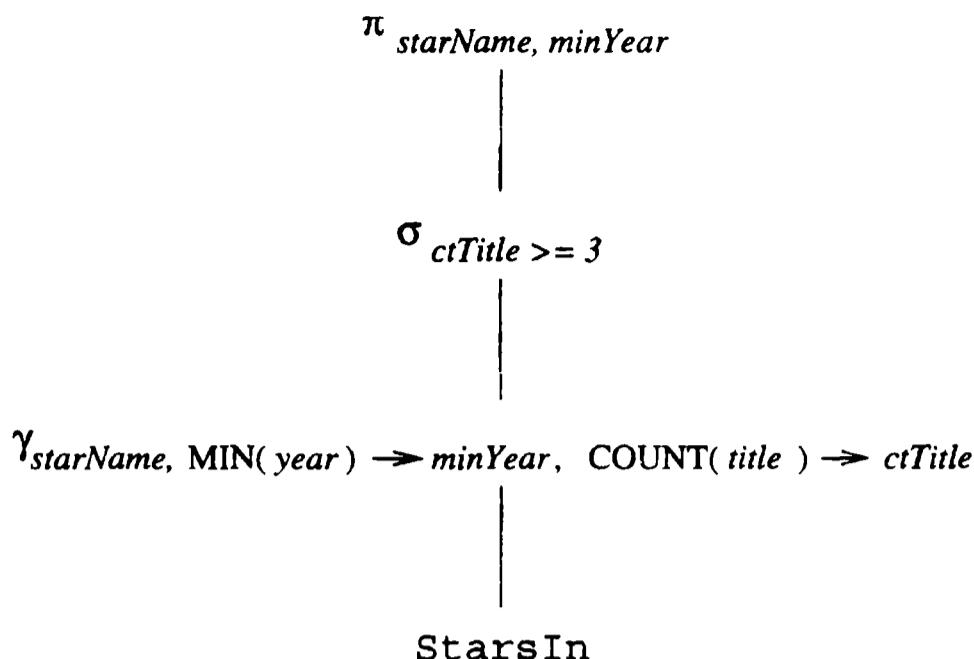


Figure 5.5: Algebraic expression tree for the query of Example 5.10

2. An expression  $x \rightarrow y$ , where  $x$  and  $y$  are names for attributes. The element  $x \rightarrow y$  in the list  $L$  asks that we take the attribute  $x$  of  $R$  and *rename* it  $y$ ; i.e., the name of this attribute in the schema of the result relation is  $y$ .
3. An expression  $E \rightarrow z$ , where  $E$  is an expression involving attributes of  $R$ , constants, arithmetic operators, and string operators, and  $z$  is a new name for the attribute that results from the calculation implied by  $E$ . For example,  $a + b \rightarrow x$  as a list element represents the sum of the attributes  $a$  and  $b$ , renamed  $x$ . Element  $c \parallel d \rightarrow e$  means concatenate the presumably string-valued attributes  $c$  and  $d$  and call the result  $e$ .

The result of the projection is computed by considering each tuple of  $R$  in turn. We evaluate the list  $L$  by substituting the tuple's components for the corresponding attributes mentioned in  $L$  and applying any operators indicated by  $L$  to these values. The result is a relation whose schema is the names of the attributes on list  $L$ , with whatever renaming the list specifies. Each tuple of  $R$  yields one tuple of the result. Duplicate tuples in  $R$  surely yield duplicate tuples in the result, but the result can have duplicates even if  $R$  does not.

**Example 5.11:** Let  $R$  be the relation

A	B	C
0	1	2
0	1	2
3	4	5

Then the result of  $\pi_{A,B+C \rightarrow X}(R)$  is

A	X
0	3
0	3
3	9

The result's schema has two attributes. One is  $A$ , the first attribute of  $R$ , not renamed. The second is the sum of the second and third attributes of  $R$ , with the name  $X$ .

For another example,  $\pi_{B-A \rightarrow X, C-B \rightarrow Y}(R)$  is

X	Y
1	1
1	1
1	1

Notice that the calculation required by this projection list happens to turn different tuples  $(0, 1, 2)$  and  $(3, 4, 5)$  into the same tuple  $(1, 1)$ . Thus, the latter tuple appears three times in the result.  $\square$

### 5.2.6 The Sorting Operator

There are several contexts in which we want to sort the tuples of a relation by one or more of its attributes. Often, when querying data, one wants the result relation to be sorted. For instance, in a query about all the movies in which Sean Connery appeared, we might wish to have the list sorted by title, so we could more easily find whether a certain movie was on the list. We shall also see when we study query optimization how execution of queries by the DBMS is often made more efficient if we sort the relations first.

The expression  $\tau_L(R)$ , where  $R$  is a relation and  $L$  a list of some of  $R$ 's attributes, is the relation  $R$ , but with the tuples of  $R$  sorted in the order indicated by  $L$ . If  $L$  is the list  $A_1, A_2, \dots, A_n$ , then the tuples of  $R$  are sorted first by their value of attribute  $A_1$ . Ties are broken according to the value of  $A_2$ ; tuples that agree on both  $A_1$  and  $A_2$  are ordered according to their value of  $A_3$ , and so on. Ties that remain after attribute  $A_n$  is considered may be ordered arbitrarily.

**Example 5.12 :** If  $R$  is a relation with schema  $R(A, B, C)$ , then  $\tau_{C,B}(R)$  orders the tuples of  $R$  by their value of  $C$ , and tuples with the same  $C$ -value are ordered by their  $B$  value. Tuples that agree on both  $B$  and  $C$  may be ordered arbitrarily.  $\square$

If we apply another operator such as join to the sorted result of a  $\tau$ , the sorted order usually becomes meaningless, and the elements on the list should be treated as a bag, not a list. However, bag projections can be made to preserve the order. Also, a selection on a list drops out the tuples that do not satisfy the condition of the selection, but the remaining tuples can be made to appear in their original sorted order.

### 5.2.7 Outerjoins

A property of the join operator is that it is possible for certain tuples to be “dangling”; that is, they fail to match any tuple of the other relation in the

common attributes. Dangling tuples do not have any trace in the result of the join, so the join may not represent the data of the original relations completely. In cases where this behavior is undesirable, a variation on the join, called “outerjoin,” has been proposed and appears in various commercial systems.

We shall consider the “natural” case first, where the join is on equated values of all attributes in common to the two relations. The *outerjoin*  $R \bowtie S$  is formed by starting with  $R \bowtie S$ , and adding any dangling tuples from  $R$  or  $S$ . The added tuples must be padded with a special *null* symbol,  $\perp$ , in all the attributes that they do not possess but that appear in the join result. Note that  $\perp$  is written `NULL` in SQL (recall Section 2.3.4).

$A$	$B$	$C$
1	2	3
4	5	6
7	8	9

(a) Relation  $U$ 

$B$	$C$	$D$
2	3	10
2	3	11
6	7	12

(b) Relation  $V$ 

$A$	$B$	$C$	$D$
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$
$\perp$	6	7	12

(c) Result  $U \bowtie V$ 

Figure 5.6: Outerjoin of relations

**Example 5.13:** In Fig. 5.6(a) and (b) we see two relations  $U$  and  $V$ . Tuple  $(1, 2, 3)$  of  $U$  joins with both  $(2, 3, 10)$  and  $(2, 3, 11)$  of  $V$ , so these three tuples are not dangling. However, the other three tuples —  $(4, 5, 6)$  and  $(7, 8, 9)$  of  $U$  and  $(6, 7, 12)$  of  $V$  — are dangling. That is, for none of these three tuples is there a tuple of the other relation that agrees with it on both the  $B$  and  $C$  components. Thus, in  $U \bowtie V$ , seen in Fig. 5.6(c), the three dangling tuples

are padded with  $\perp$  in the attributes that they do not have: attribute  $D$  for the tuples of  $U$  and attribute  $A$  for the tuple of  $V$ .  $\square$

There are many variants of the basic (natural) outerjoin idea. The *left outerjoin*  $R \bowtie_L S$  is like the outerjoin, but only dangling tuples of the left argument  $R$  are padded with  $\perp$  and added to the result. The *right outerjoin*  $R \bowtie_R S$  is like the outerjoin, but only the dangling tuples of the right argument  $S$  are padded with  $\perp$  and added to the result.

**Example 5.14:** If  $U$  and  $V$  are as in Fig. 5.6, then  $U \bowtie_L V$  is:

$A$	$B$	$C$	$D$
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$

and  $U \bowtie_R V$  is:

$A$	$B$	$C$	$D$
1	2	3	10
1	2	3	11
$\perp$	6	7	12

$\square$

In addition, all three natural outerjoin operators have theta-join analogs, where first a theta-join is taken and then those tuples that failed to join with any tuple of the other relation, when the condition of the theta-join was applied, are padded with  $\perp$  and added to the result. We use  $\bowtie_C$  to denote a theta-outerjoin with condition  $C$ . This operator can also be modified with  $L$  or  $R$  to indicate left- or right-outerjoin.

**Example 5.15:** Let  $U$  and  $V$  be the relations of Fig. 5.6, and consider

$$U \bowtie_{A>V.C} V$$

Tuples  $(4, 5, 6)$  and  $(7, 8, 9)$  of  $U$  each satisfy the condition with both of the tuples  $(2, 3, 10)$  and  $(2, 3, 11)$  of  $V$ . Thus, none of these four tuples are dangling in this theta-join. However, the two other tuples —  $(1, 2, 3)$  of  $U$  and  $(6, 7, 12)$  of  $V$  — are dangling. They thus appear, padded, in the result shown in Fig. 5.7.

$\square$

<i>A</i>	<i>U.B</i>	<i>U.C</i>	<i>V.B</i>	<i>V.C</i>	<i>D</i>
4	5	6	2	3	10
4	5	6	2	3	11
7	8	9	2	3	10
7	8	9	2	3	11
1	2	3	$\perp$	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$	6	7	12

Figure 5.7: Result of a theta-outerjoin

### 5.2.8 Exercises for Section 5.2

**Exercise 5.2.1:** Here are two relations:

$$R(A, B): \{(0, 1), (2, 3), (0, 1), (2, 4), (3, 4)\}$$

$$S(B, C): \{(0, 1), (2, 4), (2, 5), (3, 4), (0, 2), (3, 4)\}$$

Compute the following: a)  $\pi_{A+B, A^2, B^2}(R)$ ; b)  $\pi_{B+1, C-1}(S)$ ; c)  $\tau_{B, A}(R)$ ; d)  $\tau_{B, C}(S)$ ; e)  $\delta(R)$ ; f)  $\delta(S)$ ; g)  $\gamma_{A, \text{SUM}(B)}(R)$ ; h)  $\gamma_{B, \text{AVG}(C)}(S)$ ; ! i)  $\gamma_A(R)$ ; ! j)  $\gamma_{A, \text{MAX}(C)}(R \bowtie S)$ ; k)  $R \bowtie_L S$ ; l)  $R \bowtie_R S$ ; m)  $R \bowtie S$ ; n)  $R \bowtie_{R.B < S.B} S$ .

! **Exercise 5.2.2:** A unary operator  $f$  is said to be *idempotent* if for all relations  $R$ ,  $f(f(R)) = f(R)$ . That is, applying  $f$  more than once is the same as applying it once. Which of the following operators are idempotent? Either explain why or give a counterexample.

- a)  $\delta$ ; b)  $\pi_L$ ; c)  $\sigma_C$ ; d)  $\gamma_L$ ; e)  $\tau$ .

! **Exercise 5.2.3:** One thing that can be done with an extended projection, but not with the original version of projection that we defined in Section 2.4.5, is to duplicate columns. For example, if  $R(A, B)$  is a relation, then  $\pi_{A, A}(R)$  produces the tuple  $(a, a)$  for every tuple  $(a, b)$  in  $R$ . Can this operation be done using only the classical operations of relation algebra from Section 2.4? Explain your reasoning.

## 5.3 A Logic for Relations

As an alternative to abstract query languages based on algebra, one can use a form of logic to express queries. The logical query language *Datalog* (“database logic”) consists of if-then rules. Each of these rules expresses the idea that from certain combinations of tuples in certain relations, we may infer that some other tuple must be in some other relation, or in the answer to a query.

### 5.3.1 Predicates and Atoms

Relations are represented in Datalog by *predicates*. Each predicate takes a fixed number of arguments, and a predicate followed by its arguments is called an *atom*. The syntax of atoms is just like that of function calls in conventional programming languages; for example  $P(x_1, x_2, \dots, x_n)$  is an atom consisting of the predicate  $P$  with arguments  $x_1, x_2, \dots, x_n$ .

In essence, a predicate is the name of a function that returns a boolean value. If  $R$  is a relation with  $n$  attributes in some fixed order, then we shall also use  $R$  as the name of a predicate corresponding to this relation. The atom  $R(a_1, a_2, \dots, a_n)$  has value TRUE if  $(a_1, a_2, \dots, a_n)$  is a tuple of  $R$ ; the atom has value FALSE otherwise.

Notice that a relation defined by a predicate can be assumed to be a set. In Section 5.3.6, we shall discuss how it is possible to extend Datalog to bags. However, outside that section, you should assume in connection with Datalog that relations are sets.

**Example 5.16:** Let  $R$  be the relation

$A$	$B$
1	2
3	4

Then  $R(1, 2)$  is true and so is  $R(3, 4)$ . However, for any other combination of values  $x$  and  $y$ ,  $R(x, y)$  is false.  $\square$

A predicate can take variables as well as constants as arguments. If an atom has variables for one or more of its arguments, then it is a boolean-valued function that takes values for these variables and returns TRUE or FALSE.

**Example 5.17:** If  $R$  is the predicate from Example 5.16, then  $R(x, y)$  is the function that tells, for any  $x$  and  $y$ , whether the tuple  $(x, y)$  is in relation  $R$ . For the particular instance of  $R$  mentioned in Example 5.16,  $R(x, y)$  returns TRUE when either

1.  $x = 1$  and  $y = 2$ , or
2.  $x = 3$  and  $y = 4$

and returns FALSE otherwise. As another example, the atom  $R(1, z)$  returns TRUE if  $z = 2$  and returns FALSE otherwise.  $\square$

### 5.3.2 Arithmetic Atoms

There is another kind of atom that is important in Datalog: an *arithmetic atom*. This kind of atom is a comparison between two arithmetic expressions, for example  $x < y$  or  $x + 1 \geq y + 4 \times z$ . For contrast, we shall call the atoms introduced in Section 5.3.1 *relational atoms*; both kinds are “atoms.”

Note that arithmetic and relational atoms each take as arguments the values of any variables that appear in the atom, and they return a boolean value. In effect, arithmetic comparisons like  $<$  or  $\geq$  are like the names of relations that contain all the true pairs. Thus, we can visualize the relation “ $<$ ” as containing all the tuples, such as  $(1, 2)$  or  $(-1.5, 65.4)$ , whose first component is less than their second component. Remember, however, that database relations are always finite, and usually change from time to time. In contrast, arithmetic-comparison relations such as  $<$  are both infinite and unchanging.

### 5.3.3 Datalog Rules and Queries

Operations similar to those of relational algebra are described in Datalog by *rules*, which consist of

1. A relational atom called the *head*, followed by
2. The symbol  $\leftarrow$ , which we often read “if,” followed by
3. A *body* consisting of one or more atoms, called *subgoals*, which may be either relational or arithmetic. Subgoals are connected by AND, and any subgoal may optionally be preceded by the logical operator NOT.

**Example 5.18:** The Datalog rule

`LongMovie(t,y) ← Movies(t,y,l,g,s,p) AND l ≥ 100`

defines the set of “long” movies, those at least 100 minutes long. It refers to our standard relation **Movies** with schema

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

The head of the rule is the atom *LongMovie*( $t, y$ ). The body of the rule consists of two subgoals:

1. The first subgoal has predicate *Movies* and six arguments, corresponding to the six attributes of the **Movies** relation. Each of these arguments has a different variable:  $t$  for the **title** component,  $y$  for the **year** component,  $l$  for the **length** component, and so on. We can see this subgoal as saying: “Let  $(t, y, l, g, s, p)$  be a tuple in the current instance of relation **Movies**.” More precisely, *Movies*( $t, y, l, g, s, p$ ) is true whenever the six variables have values that are the six components of some one **Movies** tuple.
2. The second subgoal,  $l \geq 100$ , is true whenever the length component of a **Movies** tuple is at least 100.

The rule as a whole can be thought of as saying: *LongMovie*( $t, y$ ) is true whenever we can find a tuple in **Movies** with:

- a)  $t$  and  $y$  as the first two components (for **title** and **year**),

## Anonymous Variables

Frequently, Datalog rules have some variables that appear only once. The names used for these variables are irrelevant. Only when a variable appears more than once do we care about its name, so we can see it is the same variable in its second and subsequent appearances. Thus, we shall allow the common convention that an underscore,  $\_$ , as an argument of an atom, stands for a variable that appears only there. Multiple occurrences of  $\_$  stand for different variables, never the same variable. For instance, the rule of Example 5.18 could be written

$$\text{LongMovie}(t, y) \leftarrow \text{Movies}(t, y, l, \_, \_, \_) \text{ AND } l \geq 100$$

The three variables  $g$ ,  $s$ , and  $p$  that appear only once have each been replaced by underscores. We cannot replace any of the other variables, since each appears twice in the rule.

- b) A third component  $l$  (for `length`) that is at least 100, and
- c) Any values in components 4 through 6.

Notice that this rule is thus equivalent to the “assignment statement” in relational algebra:

$$\text{LongMovie} := \pi_{\text{title}, \text{year}}(\sigma_{\text{length} \geq 100}(\text{Movies}))$$

whose right side is a relational-algebra expression.  $\square$

A *query* in Datalog is a collection of one or more rules. If there is only one relation that appears in the rule heads, then the value of this relation is taken to be the answer to the query. Thus, in Example 5.18, `LongMovie` is the answer to the query. If there is more than one relation among the rule heads, then one of these relations is the answer to the query, while the others assist in the definition of the answer. When there are several predicates defined by a collection of rules, we shall usually assume that the query result is named `Answer`.

### 5.3.4 Meaning of Datalog Rules

Example 5.18 gave us a hint of the meaning of a Datalog rule. More precisely, imagine the variables of the rule ranging over all possible values. Whenever these variables have values that together make all the subgoals true, then we see what the value of the head is for those variables, and we add the resulting tuple to the relation whose predicate is in the head.

For instance, we can imagine the six variables of Example 5.18 ranging over all possible values. The only combinations of values that can make all the subgoals true are when the values of  $(t, y, l, g, s, p)$  in that order form a tuple of `Movies`. Moreover, since the  $l \geq 100$  subgoal must also be true, this tuple must be one where  $l$ , the value of the `length` component, is at least 100. When we find such a combination of values, we put the tuple  $(t, y)$  in the head's relation `LongMovie`.

There are, however, restrictions that we must place on the way variables are used in rules, so that the result of a rule is a finite relation and so that rules with arithmetic subgoals or with *negated* subgoals (those with NOT in front of them) make intuitive sense. This condition, which we call the *safety* condition, is:

- Every variable that appears anywhere in the rule must appear in some nonnegated, relational subgoal of the body.

In particular, any variable that appears in the head, in a negated relational subgoal, or in any arithmetic subgoal, must also appear in a nonnegated, relational subgoal of the body.

**Example 5.19:** Consider the rule

$$\text{LongMovie}(t, y) \leftarrow \text{Movies}(t, y, l, \_, \_, \_) \text{ AND } l \geq 100$$

from Example 5.18. The first subgoal is a nonnegated, relational subgoal, and it contains all the variables that appear anywhere in the rule, including the anonymous ones represented by underscores. In particular, the two variables  $t$  and  $y$  that appear in the head also appear in the first subgoal of the body. Likewise, variable  $l$  appears in an arithmetic subgoal, but it also appears in the first subgoal. Thus, the rule is safe.  $\square$

**Example 5.20:** The following rule has three safety violations:

$$P(x, y) \leftarrow Q(x, z) \text{ AND NOT } R(w, x, z) \text{ AND } x < y$$

1. The variable  $y$  appears in the head but not in any nonnegated, relational subgoal of the body. Notice that  $y$ 's appearance in the arithmetic subgoal  $x < y$  does not help to limit the possible values of  $y$  to a finite set. As soon as we find values  $a, b$ , and  $c$  for  $w, x$ , and  $z$  respectively that satisfy the first two subgoals, we are forced to add the infinite number of tuples  $(b, d)$  such that  $d > b$  to the relation for the head predicate  $P$ .
2. Variable  $w$  appears in a negated, relational subgoal but not in a non-negated, relational subgoal.
3. Variable  $y$  appears in an arithmetic subgoal, but not in a nonnegated, relational subgoal.

Thus, it is not a safe rule and cannot be used in Datalog.  $\square$

There is another way to define the meaning of rules. Instead of considering all of the possible assignments of values to variables, we consider the sets of tuples in the relations corresponding to each of the nonnegated, relational subgoals. If some assignment of tuples for each nonnegated, relational subgoal is *consistent*, in the sense that it assigns the same value to each occurrence of any one variable, then consider the resulting assignment of values to all the variables of the rule. Notice that because the rule is safe, every variable is assigned a value.

For each consistent assignment, we consider the negated, relational subgoals and the arithmetic subgoals, to see if the assignment of values to variables makes them all true. Remember that a negated subgoal is true if its atom is false. If all the subgoals are true, then we see what tuple the head becomes under this assignment of values to variables. This tuple is added to the relation whose predicate is the head.

**Example 5.21:** Consider the Datalog rule

$$P(x, y) \leftarrow Q(x, z) \text{ AND } R(z, y) \text{ AND NOT } Q(x, y)$$

Let relation  $Q$  contain the two tuples  $(1, 2)$  and  $(1, 3)$ . Let relation  $R$  contain tuples  $(2, 3)$  and  $(3, 1)$ . There are two nonnegated, relational subgoals,  $Q(x, z)$  and  $R(z, y)$ , so we must consider all combinations of assignments of tuples from relations  $Q$  and  $R$ , respectively, to these subgoals. The table of Fig. 5.8 considers all four combinations.

	Tuple for $Q(x, z)$	Tuple for $R(z, y)$	Consistent Assignment?	NOT $Q(x, y)$ True?	Resulting Head
1)	$(1, 2)$	$(2, 3)$	Yes	No	—
2)	$(1, 2)$	$(3, 1)$	No; $z = 2, 3$	Irrelevant	—
3)	$(1, 3)$	$(2, 3)$	No; $z = 3, 2$	Irrelevant	—
4)	$(1, 3)$	$(3, 1)$	Yes	Yes	$P(1, 1)$

Figure 5.8: All possible assignments of tuples to  $Q(x, z)$  and  $R(z, y)$

The second and third options in Fig. 5.8 are not consistent. Each assigns two different values to the variable  $z$ . Thus, we do not consider these tuple-assignments further.

The first option, where subgoal  $Q(x, z)$  is assigned the tuple  $(1, 2)$  and subgoal  $R(z, y)$  is assigned tuple  $(2, 3)$ , yields a consistent assignment, with  $x, y$ , and  $z$  given the values 1, 3, and 2, respectively. We thus proceed to the test of the other subgoals, those that are not nonnegated, relational subgoals. There is only one:  $\text{NOT } Q(x, y)$ . For this assignment of values to the variables, this subgoal becomes  $\text{NOT } Q(1, 3)$ . Since  $(1, 3)$  is a tuple of  $Q$ , this subgoal is false, and no head tuple is produced for the tuple-assignment (1).

The final option is (4). Here, the assignment is consistent;  $x$ ,  $y$ , and  $z$  are assigned the values 1, 1, and 3, respectively. The subgoal  $\text{NOT } Q(x, y)$  takes on the value  $\text{NOT } Q(1, 1)$ . Since  $(1, 1)$  is not a tuple of  $Q$ , this subgoal is true. We thus evaluate the head  $P(x, y)$  for this assignment of values to variables and find it is  $P(1, 1)$ . Thus the tuple  $(1, 1)$  is in the relation  $P$ . Since we have exhausted all tuple-assignments, this is the only tuple in  $P$ .  $\square$

### 5.3.5 Extensional and Intensional Predicates

It is useful to make the distinction between

- *Extensional* predicates, which are predicates whose relations are stored in a database, and
- *Intensional* predicates, whose relations are computed by applying one or more Datalog rules.

The difference is the same as that between the operands of a relational-algebra expression, which are “extensional” (i.e., defined by their *extension*, which is another name for the “current instance of a relation”) and the relations computed by a relational-algebra expression, either as the final result or as an intermediate result corresponding to some subexpression; these relations are “intensional” (i.e., defined by the programmer’s “intent”).

When talking of Datalog rules, we shall refer to the relation corresponding to a predicate as “intensional” or “extensional,” if the predicate is intensional or extensional, respectively. We shall also use the abbreviation *IDB* for “intensional database” to refer to either an intensional predicate or its corresponding relation. Similarly, we use abbreviation *EDB*, standing for “extensional database,” for extensional predicates or relations.

Thus, in Example 5.18, *Movies* is an EDB relation, defined by its extension. The predicate *Movies* is likewise an EDB predicate. Relation and predicate *LongMovie* are both intensional.

An EDB predicate can never appear in the head of a rule, although it can appear in the body of a rule. IDB predicates can appear in either the head or the body of rules, or both. It is also common to construct a single relation by using several rules with the same IDB predicate in the head. We shall see an illustration of this idea in Example 5.24, regarding the union of two relations.

By using a series of intensional predicates, we can build progressively more complicated functions of the EDB relations. The process is similar to the building of relational-algebra expressions using several operators.

### 5.3.6 Datalog Rules Applied to Bags

Datalog is inherently a logic of sets. However, as long as there are no negated, relational subgoals, the ideas for evaluating Datalog rules when relations are sets apply to bags as well. When relations are bags, it is conceptually simpler to use

the second approach for evaluating Datalog rules that we gave in Section 5.3.4. Recall this technique involves looking at each of the nonnegated, relational subgoals and substituting for it all tuples of the relation for the predicate of that subgoal. If a selection of tuples for each subgoal gives a consistent value to each variable, and the arithmetic subgoals all become true,<sup>1</sup> then we see what the head becomes with this assignment of values to variables. The resulting tuple is put in the head relation.

Since we are now dealing with bags, we do not eliminate duplicates from the head. Moreover, as we consider all combinations of tuples for the subgoals, a tuple appearing  $n$  times in the relation for a subgoal gets considered  $n$  times as the tuple for that subgoal, each time in conjunction with all combinations of tuples for the other subgoals.

**Example 5.22:** Consider the rule

$$H(x, z) \leftarrow R(x, y) \text{ AND } S(y, z)$$

where relation  $R(A, B)$  has the tuples:

A	B
1	2
1	2

and  $S(B, C)$  has tuples:

B	C
2	3
4	5
4	5

The only time we get a consistent assignment of tuples to the subgoals (i.e., an assignment where the value of  $y$  from each subgoal is the same) is when the first subgoal is assigned one of the tuples  $(1, 2)$  from  $R$  and the second subgoal is assigned tuple  $(2, 3)$  from  $S$ . Since  $(1, 2)$  appears twice in  $R$ , and  $(2, 3)$  appears once in  $S$ , there will be two assignments of tuples that give the variable assignments  $x = 1$ ,  $y = 2$ , and  $z = 3$ . The tuple of the head, which is  $(x, z)$ , is for each of these assignments  $(1, 3)$ . Thus the tuple  $(1, 3)$  appears twice in the head relation  $H$ , and no other tuple appears there. That is, the relation

1	3
1	3

---

<sup>1</sup>Note that there must not be any negated relational subgoals in the rule. There is not a clearly defined meaning of arbitrary Datalog rules with negated, relational subgoals under the bag model.

is the head relation defined by this rule. More generally, had tuple  $(1, 2)$  appeared  $n$  times in  $R$  and tuple  $(2, 3)$  appeared  $m$  times in  $S$ , then tuple  $(1, 3)$  would appear  $nm$  times in  $H$ .  $\square$

If a relation is defined by several rules, then the result is the bag-union of whatever tuples are produced by each rule.

**Example 5.23:** Consider a relation  $H$  defined by the two rules

$$\begin{aligned} H(x, y) &\leftarrow S(x, y) \text{ AND } x > 1 \\ H(x, y) &\leftarrow S(x, y) \text{ AND } y < 5 \end{aligned}$$

where relation  $S(B, C)$  is as in Example 5.22; that is,  $S = \{(2, 3), (4, 5), (4, 5)\}$ . The first rule puts each of the three tuples of  $S$  into  $H$ , since they each have a first component greater than 1. The second rule puts only the tuple  $(2, 3)$  into  $H$ , since  $(4, 5)$  does not satisfy the condition  $y < 5$ . Thus, the resulting relation  $H$  has two copies of the tuple  $(2, 3)$  and two copies of the tuple  $(4, 5)$ .  $\square$

### 5.3.7 Exercises for Section 5.3

**Exercise 5.3.1:** Write each of the queries of Exercise 2.4.1 in Datalog. You should use only safe rules, but you may wish to use several IDB predicates corresponding to subexpressions of complicated relational-algebra expressions.

**Exercise 5.3.2:** Write each of the queries of Exercise 2.4.3 in Datalog. Again, use only safe rules, but you may use several IDB predicates if you like.

**!! Exercise 5.3.3:** The requirement we gave for safety of Datalog rules is sufficient to guarantee that the head predicate has a finite relation if the predicates of the relational subgoals have finite relations. However, this requirement is too strong. Give an example of a Datalog rule that violates the condition, yet whatever finite relations we assign to the relational predicates, the head relation will be finite.

## 5.4 Relational Algebra and Datalog

Each of the relational-algebra operators of Section 2.4 can be mimicked by one or several Datalog rules. In this section we shall consider each operator in turn. We shall then consider how to combine Datalog rules to mimic complex algebraic expressions. It is also true that any single safe Datalog rule can be expressed in relational algebra, although we shall not prove that fact here. However, Datalog queries are more powerful than relational algebra when several rules are allowed to interact; they can express recursions that are not expressable in the algebra (see Example 5.35).

### 5.4.1 Boolean Operations

The boolean operations of relational algebra — union, intersection, and set difference — can each be expressed simply in Datalog. Here are the three techniques needed. We assume  $R$  and  $S$  are relations with the same number of attributes,  $n$ . We shall describe the needed rules using `Answer` as the name of the head predicate in all cases. However, we can use anything we wish for the name of the result, and in fact it is important to choose different predicates for the results of different operations.

- To take the union  $R \cup S$ , use two rules and  $n$  distinct variables

$$a_1, a_2, \dots, a_n$$

One rule has  $R(a_1, a_2, \dots, a_n)$  as the lone subgoal and the other has  $S(a_1, a_2, \dots, a_n)$  alone. Both rules have the head  $\text{Answer}(a_1, a_2, \dots, a_n)$ . As a result, each tuple from  $R$  and each tuple of  $S$  is put into the answer relation.

- To take the intersection  $R \cap S$ , use a rule with body

$$R(a_1, a_2, \dots, a_n) \text{ AND } S(a_1, a_2, \dots, a_n)$$

and head  $\text{Answer}(a_1, a_2, \dots, a_n)$ . Then, a tuple is in the answer relation if and only if it is in both  $R$  and  $S$ .

- To take the difference  $R - S$ , use a rule with body

$$R(a_1, a_2, \dots, a_n) \text{ AND NOT } S(a_1, a_2, \dots, a_n)$$

and head  $\text{Answer}(a_1, a_2, \dots, a_n)$ . Then, a tuple is in the answer relation if and only if it is in  $R$  but not in  $S$ .

**Example 5.24:** Let the schemas for the two relations be  $R(A, B, C)$  and  $S(A, B, C)$ . To avoid confusion, we use different predicates for the various results, rather than calling them all `Answer`.

To take the union  $R \cup S$  we use the two rules:

1.  $U(x, y, z) \leftarrow R(x, y, z)$
2.  $U(x, y, z) \leftarrow S(x, y, z)$

Rule (1) says that every tuple in  $R$  is a tuple in the IDB relation  $U$ . Rule (2) similarly says that every tuple in  $S$  is in  $U$ .

To compute  $R \cap S$ , we use the rule

$$I(a, b, c) \leftarrow R(a, b, c) \text{ AND } S(a, b, c)$$

Finally, the rule

$$D(a, b, c) \leftarrow R(a, b, c) \text{ AND NOT } S(a, b, c)$$

computes the difference  $R - S$ .  $\square$

## Variables Are Local to a Rule

Notice that the names we choose for variables in a rule are arbitrary and have no connection to the variables used in any other rule. The reason there is no connection is that each rule is evaluated alone and contributes tuples to its head's relation independent of other rules. Thus, for instance, we could replace the second rule of Example 5.24 by

$$U(a, b, c) \leftarrow S(a, b, c)$$

while leaving the first rule unchanged, and the two rules would still compute the union of  $R$  and  $S$ . Note, however, that when substituting one variable  $u$  for another variable  $v$  within a rule, we must substitute  $u$  for all occurrences of  $v$  within the rule. Moreover, the substituting variable  $u$  that we choose must not be a variable that already appears in the rule.

### 5.4.2 Projection

To compute a projection of a relation  $R$ , we use one rule with a single subgoal with predicate  $R$ . The arguments of this subgoal are distinct variables, one for each attribute of the relation. The head has an atom with arguments that are the variables corresponding to the attributes in the projection list, in the desired order.

**Example 5.25:** Suppose we want to project the relation

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

onto its first three attributes — `title`, `year`, and `length`. The rule

$$P(t, y, l) \leftarrow \text{Movies}(t, y, l, g, s, p)$$

serves, defining a relation called  $P$  to be the result of the projection.  $\square$

### 5.4.3 Selection

Selections can be somewhat more difficult to express in Datalog. The simple case is when the selection condition is the AND of one or more arithmetic comparisons. In that case, we create a rule with

1. One relational subgoal for the relation upon which we are performing the selection. This atom has distinct variables for each component, one for each attribute of the relation.

2. For each comparison in the selection condition, an arithmetic subgoal that is identical to this comparison. However, while in the selection condition an attribute name was used, in the arithmetic subgoal we use the corresponding variable, following the correspondence established by the relational subgoal.

**Example 5.26:** The selection

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

can be written as a Datalog rule

$$S(t, y, l, g, s, p) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } l \geq 100 \text{ AND } s = 'Fox'$$

The result is the relation  $S$ . Note that  $l$  and  $s$  are the variables corresponding to attributes `length` and `studioName` in the standard order we have used for the attributes of `Movies`.  $\square$

Now, let us consider selections that involve the **OR** of conditions. We cannot necessarily replace such selections by single Datalog rules. However, selection for the **OR** of two conditions is equivalent to selecting for each condition separately and then taking the union of the results. Thus, the **OR** of  $n$  conditions can be expressed by  $n$  rules, each of which defines the same head predicate. The  $i$ th rule performs the selection for the  $i$ th of the  $n$  conditions.

**Example 5.27:** Let us modify the selection of Example 5.26 by replacing the **AND** by an **OR** to get the selection:

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

That is, find all those movies that are either long or by Fox. We can write two rules, one for each of the two conditions:

1.  $S(t, y, l, g, s, p) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } l \geq 100$
2.  $S(t, y, l, g, s, p) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } s = 'Fox'$

Rule (1) produces movies at least 100 minutes long, and rule (2) produces movies by Fox.  $\square$

Even more complex selection conditions can be formed by several applications, in any order, of the logical operators **AND**, **OR**, and **NOT**. However, there is a widely known technique, which we shall not present here, for rearranging any such logical expression into “disjunctive normal form,” where the expression is the disjunction (**OR**) of “conjuncts.” A *conjunct*, in turn, is the **AND** of “ literals,” and a *literal* is either a comparison or a negated comparison.<sup>2</sup>

---

<sup>2</sup>See, e.g., A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1992.

We can represent any literal by a subgoal, perhaps with a NOT in front of it. If the subgoal is arithmetic, the NOT can be incorporated into the comparison operator. For example, NOT  $x \geq 100$  can be written as  $x < 100$ . Then, any conjunct can be represented by a single Datalog rule, with one subgoal for each comparison. Finally, every disjunctive-normal-form expression can be written by several Datalog rules, one rule for each conjunct. These rules take the union, or OR, of the results from each of the conjuncts.

**Example 5.28:** We gave a simple instance of this algorithm in Example 5.27. A more difficult example can be formed by negating the condition of that example. We then have the expression:

$$\sigma_{\text{NOT } (\text{length} \geq 100 \text{ OR } \text{studioName} = 'Fox'))}(\text{Movies})$$

That is, find all those movies that are neither long nor by Fox.

Here, a NOT is applied to an expression that is itself not a simple comparison. Thus, we must push the NOT down the expression, using one form of *DeMorgan's laws*, which says that the negation of an OR is the AND of the negations. That is, the selection can be rewritten:

$$\sigma_{(\text{NOT } (\text{length} \geq 100)) \text{ AND } (\text{NOT } (\text{studioName} = 'Fox'))}(\text{Movies})$$

Now, we can take the NOT's inside the comparisons to get the expression:

$$\sigma_{\text{length} < 100 \text{ AND } \text{studioName} \neq 'Fox'}(\text{Movies})$$

This expression can be converted into the Datalog rule

$$S(t, y, l, g, s, p) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } l < 100 \text{ AND } s \neq 'Fox'$$

□

**Example 5.29:** Let us consider a similar example where we have the negation of an AND in the selection. Now, we use the second form of DeMorgan's law, which says that the negation of an AND is the OR of the negations. We begin with the algebraic expression

$$\sigma_{\text{NOT } (\text{length} \geq 100 \text{ AND } \text{studioName} = 'Fox'))}(\text{Movies})$$

That is, find all those movies that are not both long and by Fox.

We apply DeMorgan's law to push the NOT below the AND, to get:

$$\sigma_{(\text{NOT } (\text{length} \geq 100)) \text{ OR } (\text{NOT } (\text{studioName} = 'Fox'))}(\text{Movies})$$

Again we take the NOT's inside the comparisons to get:

$$\sigma_{\text{length} < 100 \text{ OR } \text{studioName} \neq 'Fox'}(\text{Movies})$$

Finally, we write two rules, one for each part of the OR. The resulting Datalog rules are:

1.  $S(t, y, l, g, s, p) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } l < 100$
2.  $S(t, y, l, g, s, p) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } s \neq 'Fox'$

□

### 5.4.4 Product

The product of two relations  $R \times S$  can be expressed by a single Datalog rule. This rule has two subgoals, one for  $R$  and one for  $S$ . Each of these subgoals has distinct variables, one for each attribute of  $R$  or  $S$ . The IDB predicate in the head has as arguments all the variables that appear in either subgoal, with the variables appearing in the  $R$ -subgoal listed before those of the  $S$ -subgoal.

**Example 5.30:** Let us consider the two three-attribute relations  $R$  and  $S$  from Example 5.24. The rule

$$P(a, b, c, x, y, z) \leftarrow R(a, b, c) \text{ AND } S(x, y, z)$$

defines  $P$  to be  $R \times S$ . We have arbitrarily used variables at the beginning of the alphabet for the arguments of  $R$  and variables at the end of the alphabet for  $S$ . These variables all appear in the rule head.  $\square$

### 5.4.5 Joins

We can take the natural join of two relations by a Datalog rule that looks much like the rule for a product. The difference is that if we want  $R \bowtie S$ , then we must use the same variable for attributes of  $R$  and  $S$  that have the same name and must use different variables otherwise. For instance, we can use the attribute names themselves as the variables. The head is an IDB predicate that has each variable appearing once.

**Example 5.31:** Consider relations with schemas  $R(A, B)$  and  $S(B, C, D)$ . Their natural join may be defined by the rule

$$J(a, b, c, d) \leftarrow R(a, b) \text{ AND } S(b, c, d)$$

Notice how the variables used in the subgoals correspond in an obvious way to the attributes of the relations  $R$  and  $S$ .  $\square$

We also can convert theta-joins to Datalog. Recall from Section 2.4.12 how a theta-join can be expressed as a product followed by a selection. If the selection condition is a conjunct, that is, the AND of comparisons, then we may simply start with the Datalog rule for the product and add additional, arithmetic subgoals, one for each of the comparisons.

**Example 5.32:** Consider the relations  $U(A, B, C)$  and  $V(B, C, D)$  and the theta-join:

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

We can construct the Datalog rule

$$J(a, ub, uc, vb, vc, d) \leftarrow U(a, ub, uc) \text{ AND } V(vb, vc, d) \text{ AND } a < d \text{ AND } ub \neq vb$$

to perform the same operation. We have used  $ub$  as the variable corresponding to attribute  $B$  of  $U$ , and similarly used  $vb$ ,  $uc$ , and  $vc$ , although any six distinct variables for the six attributes of the two relations would be fine. The first two subgoals introduce the two relations, and the second two subgoals enforce the two comparisons that appear in the condition of the theta-join.  $\square$

If the condition of the theta-join is not a conjunction, then we convert it to disjunctive normal form, as discussed in Section 5.4.3. We then create one rule for each conjunct. In this rule, we begin with the subgoals for the product and then add subgoals for each literal in the conjunct. The heads of all the rules are identical and have one argument for each attribute of the two relations being theta-joined.

**Example 5.33:** In this example, we shall make a simple modification to the algebraic expression of Example 5.32. The AND will be replaced by an OR. There are no negations in this expression, so it is already in disjunctive normal form. There are two conjuncts, each with a single literal. The expression is:

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

Using the same variable-naming scheme as in Example 5.32, we obtain the two rules

1.  $J(a, ub, uc, vb, vc, d) \leftarrow U(a, ub, uc) \text{ AND } V(vb, vc, d) \text{ AND } a < d$
2.  $J(a, ub, uc, vb, vc, d) \leftarrow U(a, ub, uc) \text{ AND } V(vb, vc, d) \text{ AND } ub \neq vb$

Each rule has subgoals for the two relations involved plus a subgoal for one of the two conditions  $A < D$  or  $U.B \neq V.B$ .  $\square$

### 5.4.6 Simulating Multiple Operations with Datalog

Datalog rules are not only capable of mimicking a single operation of relational algebra. We can in fact mimic any algebraic expression. The trick is to look at the expression tree for the relational-algebra expression and create one IDB predicate for each interior node of the tree. The rule or rules for each IDB predicate is whatever we need to apply the operator at the corresponding node of the tree. Those operands of the tree that are extensional (i.e., they are relations of the database) are represented by the corresponding predicate. Operands that are themselves interior nodes are represented by the corresponding IDB predicate. The result of the algebraic expression is the relation for the predicate associated with the root of the expression tree.

**Example 5.34:** Consider the algebraic expression

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

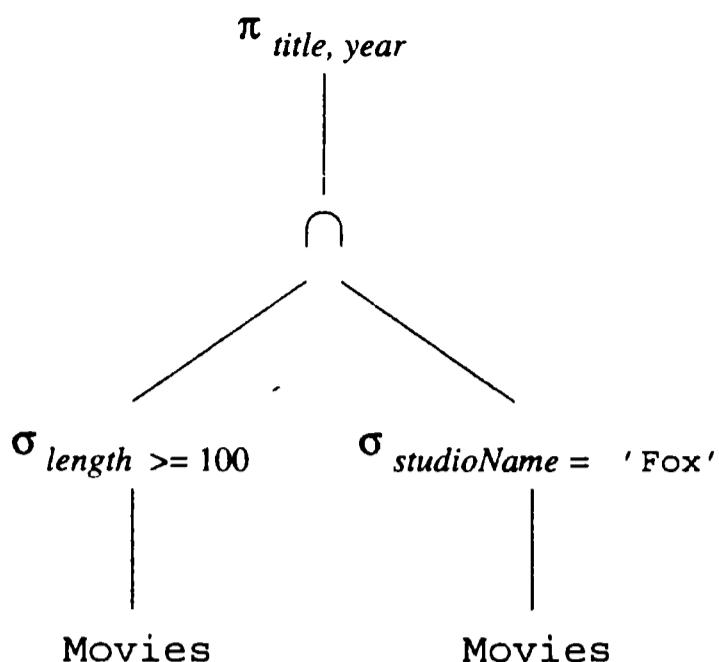


Figure 5.9: Expression tree

1.  $W(t, y, l, g, s, p) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } l \geq 100$
2.  $X(t, y, l, g, s, p) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } s = 'Fox'$
3.  $Y(t, y, l, g, s, p) \leftarrow W(t, y, l, g, s, p) \text{ AND } X(t, y, l, g, s, p)$
4.  $\text{Answer}(t, y) \leftarrow Y(t, y, l, g, s, p)$

Figure 5.10: Datalog rules to perform several algebraic operations

from Example 2.17, whose expression tree appeared in Fig. 2.18. We repeat this tree as Fig. 5.9. There are four interior nodes, so we need to create four IDB predicates. Each of these predicates has a single Datalog rule, and we summarize all the rules in Fig. 5.10.

The lowest two interior nodes perform simple selections on the EDB relation **Movies**, so we can create the IDB predicates  $W$  and  $X$  to represent these selections. Rules (1) and (2) of Fig. 5.10 describe these selections. For example, rule (1) defines  $W$  to be those tuples of **Movies** that have a length at least 100.

Then rule (3) defines predicate  $Y$  to be the intersection of  $W$  and  $X$ , using the form of rule we learned for an intersection in Section 5.4.1. Finally, rule (4) defines the answer to be the projection of  $Y$  onto the **title** and **year** attributes. We here use the technique for simulating a projection that we learned in Section 5.4.2.

Note that, because  $Y$  is defined by a single rule, we can substitute for the  $Y$  subgoal in rule (4) of Fig. 5.10, replacing it with the body of rule (3). Then, we can substitute for the  $W$  and  $X$  subgoals, using the bodies of rules (1) and (2). Since the **Movies** subgoal appears in both of these bodies, we can eliminate one copy. As a result, the single rule

$\text{Answer}(t, y) \leftarrow \text{Movies}(t, y, l, g, s, p) \text{ AND } l \geq 100 \text{ AND } s = 'Fox'$

suffices.  $\square$

### 5.4.7 Comparison Between Datalog and Relational Algebra

We see from Section 5.4.6 that every expression in the basic relational algebra of Section 2.4 can be expressed as a Datalog query. There are operations in the extended relational algebra, such as grouping and aggregation from Section 5.2, that have no corresponding features in the Datalog version we have presented here. Likewise, Datalog does not support bag operations such as duplicate elimination.

It is also true that any single Datalog rule can be expressed in relational algebra. That is, we can write a query in the basic relational algebra that produces the same set of tuples as the head of that rule produces.

However, when we consider collections of Datalog rules, the situation changes. Datalog rules can express recursion, which relational algebra can not. The reason is that IDB predicates can also be used in the bodies of rules, and the tuples we discover for the heads of rules can thus feed back to rule bodies and help produce more tuples for the heads. We shall not discuss here any of the complexities that arise, especially when the rules have negated subgoals. However, the following example will illustrate recursive Datalog.

**Example 5.35:** Suppose we have a relation  $\text{Edge}(X, Y)$  that says there is a directed edge (arc) from node  $X$  to node  $Y$ . We can express the transitive closure of the edge relation, that is, the relation  $\text{Path}(X, Y)$  meaning that there is a path of length 1 or more from node  $X$  to node  $Y$ , as follows:

1.  $\text{Path}(X, Y) \leftarrow \text{Edge}(X, Y)$
2.  $\text{Path}(X, Y) \leftarrow \text{Edge}(X, Z) \text{ AND } \text{Path}(Z, Y)$

Rule (1) says that every edge is a path. Rule (2) says that if there is an edge from node  $X$  to some node  $Z$  and a path from  $Z$  to  $Y$ , then there is also a path from  $X$  to  $Y$ . If we apply Rule (1) and then Rule (2), we get the paths of length 2. If we take the  $\text{Path}$  facts we get from this application and use them in another application of Rule (2), we get paths of length 3. Feeding those  $\text{Path}$  facts back again gives us paths of length 4, and so on. Eventually, we discover all possible path facts, and on one round we get no new facts. At that point, we can stop. If we haven't discovered the fact  $\text{Path}(a, b)$ , then there really is no path in the graph from node  $a$  to node  $b$ .  $\square$

### 5.4.8 Exercises for Section 5.4

**Exercise 5.4.1:** Let  $R(a, b, c)$ ,  $S(a, b, c)$ , and  $T(a, b, c)$  be three relations. Write one or more Datalog rules that define the result of each of the following expressions of relational algebra:

- a)  $R \cup S$ .
- b)  $R \cap S$ .

- c)  $R - S$ .
- d)  $(R \cup S) - T$ .
- ! e)  $(R - S) \cap (R - T)$ .
- f)  $\pi_{a,b}(R)$ .
- ! g)  $\pi_{a,b}(R) \cap \rho_{U(a,b)}(\pi_{b,c}(S))$ .

**Exercise 5.4.2:** Let  $R(x, y, z)$  be a relation. Write one or more Datalog rules that define  $\sigma_C(R)$ , where  $C$  stands for each of the following conditions:

- a)  $x = y$ .
- b)  $x < y \text{ AND } y < z$ .
- c)  $x < y \text{ OR } y < z$ .
- d)  $\text{NOT } (x < y \text{ OR } x > y)$ .
- ! e)  $\text{NOT } ((x < y \text{ OR } x > y) \text{ AND } y < z)$ .
- ! f)  $\text{NOT } ((x < y \text{ OR } x < z) \text{ AND } y < z)$ .

**Exercise 5.4.3:** Let  $R(a, b, c)$ ,  $S(b, c, d)$ , and  $T(d, e)$  be three relations. Write single Datalog rules for each of the natural joins:

- a)  $R \bowtie S$ .
- b)  $S \bowtie T$ .
- c)  $(R \bowtie S) \bowtie T$ . (Note: since the natural join is associative and commutative, the order of the join of these three relations is irrelevant.)

**Exercise 5.4.4:** Let  $R(x, y, z)$  and  $S(x, y, z)$  be two relations. Write one or more Datalog rules to define each of the theta-joins  $R \bowtie_C S$ , where  $C$  is one of the conditions of Exercise 5.4.2. For each of these conditions, interpret each arithmetic comparison as comparing an attribute of  $R$  on the left with an attribute of  $S$  on the right. For instance,  $x < y$  stands for  $R.x < S.y$ .

! **Exercise 5.4.5:** It is also possible to convert Datalog rules into equivalent relational-algebra expressions. While we have not discussed the method of doing so in general, it is possible to work out many simple examples. For each of the Datalog rules below, write an expression of relational algebra that defines the same relation as the head of the rule.

- a)  $P(x, y) \leftarrow Q(x, z) \text{ AND } R(z, y)$
- b)  $P(x, y) \leftarrow Q(x, z) \text{ AND } Q(z, y)$
- c)  $P(x, y) \leftarrow Q(x, z) \text{ AND } R(z, y) \text{ AND } x < y$

## 5.5 Summary of Chapter 5

- ◆ *Relations as Bags*: In commercial database systems, relations are actually bags, in which the same tuple is allowed to appear several times. The operations of relational algebra on sets can be extended to bags, but there are some algebraic laws that fail to hold.
- ◆ *Extensions to Relational Algebra*: To match the capabilities of SQL, some operators not present in the core relational algebra are needed. Sorting of a relation is an example, as is an extended projection, where computation on columns of a relation is supported. Grouping, aggregation, and outerjoins are also needed.
- ◆ *Grouping and Aggregation*: Aggregations summarize a column of a relation. Typical aggregation operators are sum, average, count, minimum, and maximum. The grouping operator allows us to partition the tuples of a relation according to their value(s) in one or more attributes before computing aggregation(s) for each group.
- ◆ *Outerjoins*: The outerjoin of two relations starts with a join of those relations. Then, dangling tuples (those that failed to join with any tuple) from either relation are padded with null values for the attributes belonging only to the other relation, and the padded tuples are included in the result.
- ◆ *Datalog*: This form of logic allows us to write queries in the relational model. In Datalog, one writes rules in which a head predicate or relation is defined in terms of a body, consisting of subgoals.
- ◆ *Atoms*: The head and subgoals are each atoms, and an atom consists of an (optionally negated) predicate applied to some number of arguments. Predicates may represent either relations or arithmetic comparisons such as  $<$ .
- ◆ *IDB and EDB Predicates*: Some predicates correspond to stored relations, and are called EDB (extensional database) predicates or relations. Other predicates, called IDB (intensional database), are defined by the rules. EDB predicates may not appear in rule heads.
- ◆ *Safe Rules*: Datalog rules must be safe, meaning that every variable in the rule appears in some nonnegated, relational subgoal of the body. Safe rules guarantee that if the EDB relations are finite, then the IDB relations will be finite.
- ◆ *Relational Algebra and Datalog*: All queries that can be expressed in core relational algebra can also be expressed in Datalog. If the rules are safe and nonrecursive, then they define exactly the same set of queries as core relational algebra.

## 5.6 References for Chapter 5

As mentioned in Chapter 2, the relational algebra comes from [2]. The extended operator  $\gamma$  is from [5].

Codd also introduced two forms of first-order logic called *tuple relational calculus* and *domain relational calculus* in one of his early papers on the relational model [3]. These forms of logic are equivalent in expressive power to relational algebra, a fact proved in [3].

Datalog, looking more like logical rules, was inspired by the programming language Prolog. The book [4] originated much of the development of logic as a query language, while [1] placed the ideas in the context of database systems.

More on Datalog and relational calculus can be found in [6] and [7].

1. F. Bancilhon, and R. Ramakrishnan, “An amateur’s introduction to recursive query-processing strategies,” *ACM SIGMOD Intl. Conf. on Management of Data*, pp. 16–52, 1986.
2. E. F. Codd, “A relational model for large shared data banks,” *Comm. ACM* **13**:6, pp. 377–387, 1970.
3. E. F. Codd, “Relational completeness of database sublanguages,” in *Database Systems* (R. Rustin, ed.), Prentice Hall, Englewood Cliffs, NJ, 1972.
4. H. Gallaire and J. Minker, *Logic and Databases*, Plenum Press, New York, 1978.
5. A. Gupta, V. Harinarayan, and D. Quass, “Generalized projections: a powerful approach to aggregation,” 21st Intl. Conf. on Very Large Database Systems, pp. 358–369, 1995.
6. M. Liu, “Deductive database languages: problems and solutions,” *Computing Surveys* **31**:1 (March, 1999), pp. 27–62.
7. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volumes I and II*, Computer Science Press, New York, 1988, 1989.



# Chapter 6

## The Database Language SQL

The most commonly used relational DBMS's query and modify the database through a language called SQL (sometimes pronounced "sequel"). SQL stands for "Structured Query Language." The portion of SQL that supports queries has capabilities very close to that of relational algebra, as extended in Section 5.2. However, SQL also includes statements for modifying the database (e.g., inserting and deleting tuples from relations) and for declaring a database schema. Thus, SQL serves as both a data-manipulation language and as a data-definition language. SQL also standardizes many other database commands, covered in Chapters 7 and 9.

There are many different dialects of SQL. First, there are three major standards. There is ANSI (American National Standards Institute) SQL and an updated standard adopted in 1992, called SQL-92 or SQL2. The most recent SQL-99 (previously referred to as SQL3) standard extends SQL2 with object-relational features and a number of other new capabilities. There is also a collection of extensions to SQL-99, collectively called SQL:2003. Then, there are versions of SQL produced by the principal DBMS vendors. These all include the capabilities of the original ANSI standard. They also conform to a large extent to the more recent SQL2, although each has its variations and extensions beyond SQL2, including some, but not all, of the features in the SQL-99 and SQL:2003 standards.

This chapter introduces the basics of SQL: the query language and database modification statements. We also introduce the notion of a "transaction," the basic unit of work for database systems. This study, although simplified, will give you a sense of how database operations can interact and some of the resulting pitfalls.

The next chapter discusses constraints and triggers, as another way of exerting user control over the content of the database. Chapter 8 covers some of the ways that we can make our SQL queries more efficient, principally by

declaring indexes and related structures. Chapter 9 covers database-related programming as part of a whole system, such as the servers that we commonly access over the Web. There, we shall see that SQL queries and other operations are almost never performed in isolation, but are embedded in a conventional host language, with which it must interact.

Finally, Chapter 10 explains a number of advanced database programming concepts. These include recursive SQL, security and access control in SQL, object-relational SQL, and the data-cube model of data.

The intent of this chapter and the following are to provide the reader with a sense of what SQL is about, more at the level of a “tutorial” than a “manual.” Thus, we focus on the most commonly used features only, and we try to use code that not only conforms to the standard, but to the usage of commercial DBMS’s. The references mention places where more of the details of the language and its dialects can be found.

## 6.1 Simple Queries in SQL

Perhaps the simplest form of query in SQL asks for those tuples of some one relation that satisfy a condition. Such a query is analogous to a selection in relational algebra. This simple query, like almost all SQL queries, uses the three keywords, **SELECT**, **FROM**, and **WHERE** that characterize SQL.

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

Figure 6.1: Example database schema, repeated

**Example 6.1:** In this and subsequent examples, we shall use the movie database schema from Section 2.2.8. For reference, these relation schemas are the ones shown in Fig. 6.1.

As our first query, let us ask about the relation

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

for all movies produced by Disney Studios in 1990. In SQL, we say

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

This query exhibits the characteristic select-from-where form of most SQL queries.

## How SQL is Used

In this chapter, we assume a *generic query interface*, where we type SQL queries or other statements and have them execute. In practice, the generic interface is used rarely. Rather, there are large programs, written in a conventional language such as C or Java (called the *host language*). These programs issue SQL statements to a database, using a special library for the host language. Data is moved from host-language variables to the SQL statements, and the results of those statements are moved from the database to host-language variables. We shall have more to say about the matter in Chapter 9.

- The **FROM** clause gives the relation or relations to which the query refers. In our example, the query is about the relation **Movies**.
- The **WHERE** clause is a condition, much like a selection-condition in relational algebra. Tuples must satisfy the condition in order to match the query. Here, the condition is that the **studioName** attribute of the tuple has the value '**Disney**' and the **year** attribute of the tuple has the value 1990. All tuples meeting both stipulations satisfy the condition; other tuples do not.
- The **SELECT** clause tells which attributes of the tuples matching the condition are produced as part of the answer. The **\*** in this example indicates that the entire tuple is produced. The result of the query is the relation consisting of all tuples produced by this process.

One way to interpret this query is to consider each tuple of the relation mentioned in the **FROM** clause. The condition in the **WHERE** clause is applied to the tuple. More precisely, any attributes mentioned in the **WHERE** clause are replaced by the value in the tuple's component for that attribute. The condition is then evaluated, and if true, the components appearing in the **SELECT** clause are produced as one tuple of the answer. Thus, the result of the query is the **Movies** tuples for those movies produced by Disney in 1990, for example, *Pretty Woman*.

In detail, when the SQL query processor encounters the **Movies** tuple

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>producerC#</i>
Pretty Woman	1990	119	true	Disney	999

(here, 999 is the imaginary certificate number for the producer of the movie), the value '**Disney**' is substituted for attribute **studioName** and value 1990 is substituted for attribute **year** in the condition of the **WHERE** clause, because these are the values for those attributes in the tuple in question. The **WHERE** clause thus becomes

## A Trick for Reading and Writing Queries

It is generally easiest to examine a select-from-where query by first looking at the **FROM** clause, to learn which relations are involved in the query. Then, move to the **WHERE** clause, to learn what it is about tuples that is important to the query. Finally, look at the **SELECT** clause to see what the output is. The same order — from, then where, then select — is often useful when writing queries of your own, as well.

**WHERE** 'Disney' = 'Disney' AND 1990 = 1990

Since this condition is evidently true, the tuple for *Pretty Woman* passes the test of the **WHERE** clause and the tuple becomes part of the result of the query.

□

### 6.1.1 Projection in SQL

We can, if we wish, eliminate some of the components of the chosen tuples; that is, we can project the relation produced by a SQL query onto some of its attributes. In place of the \* of the **SELECT** clause, we may list some of the attributes of the relation mentioned in the **FROM** clause. The result will be projected onto the attributes listed.<sup>1</sup>

**Example 6.2:** Suppose we wish to modify the query of Example 6.1 to produce only the movie title and length. We may write

```
SELECT title, length
  FROM Movies
 WHERE studioName = 'Disney' AND year = 1990;
```

The result is a table with two columns, headed **title** and **length**. The tuples in this table are pairs, each consisting of a movie title and its length, such that the movie was produced by Disney in 1990. For instance, the relation schema and one of its tuples looks like:

<i>title</i>	<i>length</i>
Pretty Woman	119
...	...

□

---

<sup>1</sup>Thus, the keyword **SELECT** in SQL actually corresponds most closely to the projection operator of relational algebra, while the selection operator of the algebra corresponds to the **WHERE** clause of SQL queries.

Sometimes, we wish to produce a relation with column headers different from the attributes of the relation mentioned in the **FROM** clause. We may follow the name of the attribute by the keyword **AS** and an *alias*, which becomes the header in the result relation. Keyword **AS** is optional. That is, an alias can immediately follow what it stands for, without any intervening punctuation.

**Example 6.3:** We can modify Example 6.2 to produce a relation with attributes **name** and **duration** in place of **title** and **length** as follows.

```
SELECT title AS name, length AS duration
  FROM Movies
 WHERE studioName = 'Disney' AND year = 1990;
```

The result is the same set of tuples as in Example 6.2, but with the columns headed by attributes **name** and **duration**. For example,

<i>name</i>	<i>duration</i>
Pretty Woman	119
...	...

could be the first tuple in the result.  $\square$

Another option in the **SELECT** clause is to use an expression in place of an attribute. Put another way, the **SELECT** list can function like the lists in an extended projection, which we discussed in Section 5.2.5. We shall see in Section 6.4 that the **SELECT** list can also include aggregates as in the  $\gamma$  operator of Section 5.2.4.

**Example 6.4:** Suppose we want output as in Example 6.3, but with the length in hours. We might replace the **SELECT** clause of that example with

```
SELECT title AS name, length*0.016667 AS lengthInHours
```

Then the same movies would be produced, but lengths would be calculated in hours and the second column would be headed by attribute **lengthInHours**, as:

<i>name</i>	<i>lengthInHours</i>
Pretty Woman	1.98334
...	...

$\square$

**Example 6.5:** We can even allow a constant as an expression in the **SELECT** clause. It might seem pointless to do so, but one application is to put some useful words into the output that SQL displays. The following query:

## Case Insensitivity

SQL is *case insensitive*, meaning that it treats upper- and lower-case letters as the same letter. For example, although we have chosen to write keywords like `FROM` in capitals, it is equally proper to write this keyword as `From` or `from`, or even `Fr0m`. Names of attributes, relations, aliases, and so on are similarly case insensitive. Only inside quotes does SQL make a distinction between upper- and lower-case letters. Thus, '`FROM`' and '`from`' are different character strings. Of course, neither is the keyword `FROM`.

```
SELECT title, length*0.016667 AS length, 'hrs.' AS inHours
  FROM Movies
 WHERE studioName = 'Disney' AND year = 1990;
```

produces tuples such as

<i>title</i>	<i>length</i>	<i>inHours</i>
Pretty Woman	1.98334	hrs.
...	...	...

We have arranged that the third column is called `inHours`, which fits with the column header `length` in the second column. Every tuple in the answer will have the constant `hrs.` in the third column, which gives the illusion of being the units attached to the value in the second column. □

### 6.1.2 Selection in SQL

The selection operator of relational algebra, and much more, is available through the `WHERE` clause of SQL. The expressions that may follow `WHERE` include conditional expressions like those found in common languages such as C or Java.

We may build expressions by comparing values using the six common comparison operators: `=`, `<>`, `<`, `>`, `<=`, and `>=`. The last four operators are as in C, but `<>` is the SQL symbol for “not equal to” (`!=` in C), and `=` in SQL is equality (`==` in C).

The values that may be compared include constants and attributes of the relations mentioned after `FROM`. We may also apply the usual arithmetic operators, `+`, `*`, and so on, to numeric values before we compare them. For instance, `(year - 1930) * (year - 1930) < 100` is true for those years within 9 of 1930. We may apply the concatenation operator `||` to strings; for example `'foo' || 'bar'` has value `'foobar'`.

An example comparison is

```
studioName = 'Disney'
```

## SQL Queries and Relational Algebra

The simple SQL queries that we have seen so far all have the form:

```
SELECT L
FROM R
WHERE C
```

in which  $L$  is a list of expressions,  $R$  is a relation, and  $C$  is a condition. The meaning of any such expression is the same as that of the relational-algebra expression

$$\pi_L(\sigma_C(R))$$

That is, we start with the relation in the `FROM` clause, apply to each tuple whatever condition is indicated in the `WHERE` clause, and then project onto the list of attributes and/or expressions in the `SELECT` clause.

in Example 6.1. The attribute `studioName` of the relation `Movies` is tested for equality against the constant '`Disney`'. This constant is string-valued; strings in SQL are denoted by surrounding them with single quotes. Numeric constants, integers and reals, are also allowed, and SQL uses the common notations for reals such as `-12.34` or `1.23E45`.

The result of a comparison is a boolean value: either `TRUE` or `FALSE`.<sup>2</sup> Boolean values may be combined by the logical operators `AND`, `OR`, and `NOT`, with their expected meanings. For instance, we saw in Example 6.1 how two conditions could be combined by `AND`. The `WHERE` clause of this example evaluates to true if and only if both comparisons are satisfied; that is, the studio name is '`Disney`' and the year is 1990. Here is an example of a query with a complex `WHERE` clause.

**Example 6.6:** Consider the query

```
SELECT title
FROM Movies
WHERE (year > 1970 OR length < 90) AND studioName = 'MGM';
```

This query asks for the titles of movies made by MGM Studios that either were made after 1970 or were less than 90 minutes long. Notice that comparisons can be grouped using parentheses. The parentheses are needed here because the precedence of logical operators in SQL is the same as in most other languages: `AND` takes precedence over `OR`, and `NOT` takes precedence over both. □

---

<sup>2</sup>Well there's a bit more to boolean values; see Section 6.1.7.

## Representing Bit Strings

A string of bits is represented by B followed by a quoted string of 0's and 1's. Thus, B'011' represents the string of three bits, the first of which is 0 and the other two of which are 1. Hexadecimal notation may also be used, where an X is followed by a quoted string of hexadecimal digits (0 through 9, and a through f, with the latter representing “digits” 10 through 15). For instance, X'7ff' represents a string of twelve bits, a 0 followed by eleven 1's. Note that each hexadecimal digit represents four bits, and leading 0's are not suppressed.

### 6.1.3 Comparison of Strings

Two strings are equal if they are the same sequence of characters. Recall from Section 2.3.2 that strings can be stored as fixed-length strings, using CHAR, or variable-length strings, using VARCHAR. When comparing strings with different declarations, only the actual strings are compared; SQL ignores any “pad” characters that must be present in the database in order to give a string its required length.

When we compare strings by one of the “less than” operators, such as < or >=, we are asking whether one precedes the other in lexicographic order (i.e., in dictionary order, or alphabetically). That is, if  $a_1a_2 \dots a_n$  and  $b_1b_2 \dots b_m$  are two strings, then the first is “less than” the second if either  $a_1 < b_1$ , or if  $a_1 = b_1$  and  $a_2 < b_2$ , or if  $a_1 = b_1$ ,  $a_2 = b_2$ , and  $a_3 < b_3$ , and so on. We also say  $a_1a_2 \dots a_n < b_1b_2 \dots b_m$  if  $n < m$  and  $a_1a_2 \dots a_n = b_1b_2 \dots b_n$ ; that is, the first string is a proper prefix of the second. For instance, 'fodder' < 'foo', because the first two characters of each string are the same, fo, and the third character of **fodder** precedes the third character of **foo**. Also, 'bar' < 'bargain' because the former is a proper prefix of the latter.

### 6.1.4 Pattern Matching in SQL

SQL also provides the capability to compare strings on the basis of a simple pattern match. An alternative form of comparison expression is

**s LIKE p**

where *s* is a string and *p* is a *pattern*, that is, a string with the optional use of the two special characters % and \_\_. Ordinary characters in *p* match only themselves in *s*. But % in *p* can match any sequence of 0 or more characters in *s*, and \_ in *p* matches any one character in *s*. The value of this expression is true if and only if string *s* matches pattern *p*. Similarly, *s NOT LIKE p* is true if and only if string *s* does not match pattern *p*.

**Example 6.7:** We remember a movie “Star something,” and we remember that the something has four letters. What could this movie be? We can retrieve all such names with the query:

```
SELECT title
  FROM Movies
 WHERE title LIKE 'Star ____';
```

This query asks if the title attribute of a movie has a value that is nine characters long, the first five characters being **Star** and a blank. The last four characters may be anything, since any sequence of four characters matches the four **\_** symbols. The result of the query is the set of complete matching titles, such as *Star Wars* and *Star Trek*. □

**Example 6.8:** Let us search for all movies with a possessive ('s) in their titles. The desired query is

```
SELECT title
  FROM Movies
 WHERE title LIKE ''''s''';
```

To understand this pattern, we must first observe that the apostrophe, being the character that surrounds strings in SQL, cannot also represent itself. The convention taken by SQL is that two consecutive apostrophes in a string represent a single apostrophe and do not end the string. Thus, ''s in a pattern is matched by a single apostrophe followed by an s.

The two % characters on either side of the 's match any strings whatsoever. Thus, any title with 's as a substring will match the pattern, and the answer to this query will include films such as *Logan's Run* or *Alice's Restaurant*. □

### 6.1.5 Dates and Times

Implementations of SQL generally support dates and times as special data types. These values are often representable in a variety of formats such as 05/14/1948 or 14 May 1948. Here we shall describe only the SQL standard notation, which is very specific about format.

A *date* constant is represented by 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. Note that, as in our example, a one-digit month is padded with a leading 0. Finally there is another hyphen and two digits representing the day. As with months, we pad the day with a leading 0 if that is necessary to make a two-digit number.

A *time* constant is represented similarly by the keyword TIME and a quoted string. This string has two digits for the hour, on the military (24-hour)

## Escape Characters in LIKE expressions

What if the pattern we wish to use in a LIKE expression involves the characters % or \_? Instead of having a particular character used as the escape character (e.g., the backslash in most UNIX commands), SQL allows us to specify any one character we like as the escape character for a single pattern. We do so by following the pattern by the keyword ESCAPE and the chosen escape character, in quotes. A character % or \_ preceded by the escape character in the pattern is interpreted literally as that character, not as a symbol for any sequence of characters or any one character, respectively. For example,

```
s LIKE 'x%/%x%' ESCAPE 'x'
```

makes x the escape character in the pattern x%/%x%. The sequence x% is taken to be a single %. This pattern matches any string that begins and ends with the character %. Note that only the middle % has its “any string” interpretation.

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.

Alternatively, time can be expressed as the number of hours and minutes ahead of (indicated by a plus sign) or behind (indicated by a minus sign) Greenwich Mean Time (GMT). For instance, TIME '12:00:00-8:00' represents noon in Pacific Standard Time, which is eight hours behind GMT.

To combine dates and times we use a value of type TIMESTAMP. These values consist of the keyword TIMESTAMP, a date value, a space, and a time value. Thus, TIMESTAMP '1948-05-14 12:00:00' represents noon on May 14, 1948.

We can compare dates or times using the same comparison operators we use for numbers or strings. That is, < on dates means that the first date is earlier than the second; < on times means that the first is earlier (within the same day) than the second.

### 6.1.6 Null Values and Comparisons Involving NULL

SQL allows attributes to have a special value NULL, which is called the *null value*. There are many different interpretations that can be put on null values. Here are some of the most common:

1. *Value unknown*: that is, “I know there is some value that belongs here but I don’t know what it is.” An unknown birthdate is an example.

2. *Value inapplicable*: “There is no value that makes sense here.” For example, if we had a `spouse` attribute for the `MovieStar` relation, then an unmarried star might have `NULL` for that attribute, not because we don’t know the spouse’s name, but because there is none.
3. *Value withheld*: “We are not entitled to know the value that belongs here.” For instance, an unlisted phone number might appear as `NULL` in the component for a `phone` attribute.

We saw in Section 5.2.7 how the use of the `outerjoin` operator of relational algebra produces null values in some components of tuples; SQL allows outerjoins and also produces `NULL`’s when a query involves outerjoins; see Section 6.3.8. There are other ways SQL produces `NULL`’s as well. For example, certain insertions of tuples create null values, as we shall see in Section 6.5.1.

In `WHERE` clauses, we must be prepared for the possibility that a component of some tuple we are examining will be `NULL`. There are two important rules to remember when we operate upon a `NULL` value.

1. When we operate on a `NULL` and any value, including another `NULL`, using an arithmetic operator like `*` or `+`, the result is `NULL`.
2. When we compare a `NULL` value and any value, including another `NULL`, using a comparison operator like `=` or `>`, the result is `UNKNOWN`. The value `UNKNOWN` is another truth-value, like `TRUE` and `FALSE`; we shall discuss how to manipulate truth-value `UNKNOWN` shortly.

However, we must remember that, although `NULL` is a value that can appear in tuples, it is *not* a constant. Thus, while the above rules apply when we try to operate on an expression whose value is `NULL`, we cannot use `NULL` explicitly as an operand.

**Example 6.9:** Let  $x$  have the value `NULL`. Then the value of  $x + 3$  is also `NULL`. However, `NULL + 3` is not a legal SQL expression. Similarly, the value of  $x = 3$  is `UNKNOWN`, because we cannot tell if the value of  $x$ , which is `NULL`, equals the value 3. However, the comparison `NULL = 3` is not correct SQL.  $\square$

The correct way to ask if  $x$  has the value `NULL` is with the expression `x IS NULL`. This expression has the value `TRUE` if  $x$  has the value `NULL` and it has value `FALSE` otherwise. Similarly `x IS NOT NULL` has the value `TRUE` unless the value of  $x$  is `NULL`.

### 6.1.7 · The Truth-Value UNKNOWN

In Section 6.1.2 we assumed that the result of a comparison was either `TRUE` or `FALSE`, and these truth-values were combined in the obvious way using the logical operators `AND`, `OR`, and `NOT`. We have just seen that when `NULL` values

## Pitfalls Regarding Nulls

It is tempting to assume that `NULL` in SQL can always be taken to mean “a value that we don’t know but that surely exists.” However, there are several ways that intuition is violated. For instance, suppose  $x$  is a component of some tuple, and the domain for that component is the integers. We might reason that  $0 * x$  surely has the value 0, since no matter what integer  $x$  is, its product with 0 is 0. However, if  $x$  has the value `NULL`, rule (1) of Section 6.1.6 applies; the product of 0 and `NULL` is `NULL`. Similarly, we might reason that  $x - x$  has the value 0, since whatever integer  $x$  is, its difference with itself is 0. However, again the rule about operations on nulls applies, and the result is `NULL`.

occur, comparisons can yield a third truth-value: `UNKNOWN`. We must now learn how the logical operators behave on combinations of all three truth-values.

The rule is easy to remember if we think of `TRUE` as 1 (i.e., fully true), `FALSE` as 0 (i.e., not at all true), and `UNKNOWN` as 1/2 (i.e., somewhere between true and false). Then:

1. The `AND` of two truth-values is the minimum of those values. That is,  $x \text{ AND } y$  is `FALSE` if either  $x$  or  $y$  is `FALSE`; it is `UNKNOWN` if neither is `FALSE` but at least one is `UNKNOWN`, and it is `TRUE` only when both  $x$  and  $y$  are `TRUE`.
2. The `OR` of two truth-values is the maximum of those values. That is,  $x \text{ OR } y$  is `TRUE` if either  $x$  or  $y$  is `TRUE`; it is `UNKNOWN` if neither is `TRUE` but at least one is `UNKNOWN`, and it is `FALSE` only when both are `FALSE`.
3. The negation of truth-value  $v$  is  $1 - v$ . That is, `NOT`  $x$  has the value `TRUE` when  $x$  is `FALSE`, the value `FALSE` when  $x$  is `TRUE`, and the value `UNKNOWN` when  $x$  has value `UNKNOWN`.

In Fig. 6.2 is a summary of the result of applying the three logical operators to the nine different combinations of truth-values for operands  $x$  and  $y$ . The value of the last operator, `NOT`, depends only on  $x$ .

SQL conditions, as appear in `WHERE` clauses of select-from-where statements, apply to each tuple in some relation, and for each tuple, one of the three truth values, `TRUE`, `FALSE`, or `UNKNOWN` is produced. However, only the tuples for which the condition has the value `TRUE` become part of the answer; tuples with either `UNKNOWN` or `FALSE` as value are excluded from the answer. That situation leads to another surprising behavior similar to that discussed in the box on “Pitfalls Regarding Nulls,” as the next example illustrates.

**Example 6.10:** Suppose we ask about our running-example relation

<i>x</i>	<i>y</i>	<i>x AND y</i>	<i>x OR y</i>	<i>NOT x</i>
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Figure 6.2: Truth table for three-valued logic

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

the following query:

```
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120;
```

Intuitively, we would expect to get a copy of the **Movies** relation, since each movie has a length that is either 120 or less or that is greater than 120.

However, suppose there are **Movies** tuples with NULL in the **length** component. Then both comparisons **length**  $\leq$  120 and **length**  $>$  120 evaluate to UNKNOWN. The OR of two UNKNOWN's is UNKNOWN, by Fig. 6.2. Thus, for any tuple with a NULL in the **length** component, the WHERE clause evaluates to UNKNOWN. Such a tuple is *not* returned as part of the answer to the query. As a result, the true meaning of the query is “find all the **Movies** tuples with non-NUL lengths.” □

### 6.1.8 Ordering the Output

We may ask that the tuples produced by a query be presented in sorted order. The order may be based on the value of any attribute, with ties broken by the value of a second attribute, remaining ties broken by a third, and so on, as in the  $\tau$  operation of Section 5.2.6. To get output in sorted order, we may add to the select-from-where statement a clause:

```
ORDER BY <list of attributes>
```

The order is by default ascending, but we can get the output highest-first by appending the keyword DESC (for “descending”) to an attribute. Similarly, we can specify ascending order with the keyword ASC, but that word is unnecessary.

The ORDER BY clause follows the WHERE clause and any other clauses (i.e., the optional GROUP BY and HAVING clauses, which are introduced in Section 6.4).

The ordering is performed on the result of the **FROM**, **WHERE**, and other clauses, just before we apply the **SELECT** clause. The tuples of this result are then sorted by the attributes in the list of the **ORDER BY** clause, and then passed to the **SELECT** clause for processing in the normal manner.

**Example 6.11:** The following is a rewrite of our original query of Example 6.1, asking for the Disney movies of 1990 from the relation

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

To get the movies listed by length, shortest first, and among movies of equal length, alphabetically, we can say:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

A subtlety of ordering is that all the attributes of **Movies** are available at the time of sorting, even if they are not part of the **SELECT** clause. Thus, we could replace **SELECT \*** by **SELECT producerC#**, and the query would still be legal.

□

An additional option in ordering is that the list following **ORDER BY** can include expressions, just as the **SELECT** clause can. For instance, we can order the tuples of a relation  $R(A, B)$  by the sum of the two components of the tuples, highest first, with:

```
SELECT *
FROM R
ORDER BY A+B DESC;
```

### 6.1.9 Exercises for Section 6.1

**Exercise 6.1.1:** If a query has a **SELECT** clause

```
SELECT A B
```

how do we know whether  $A$  and  $B$  are two different attributes or  $B$  is an alias of  $A$ ?

**Exercise 6.1.2:** Write the following queries, based on our running movie database example

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

in SQL.

- a) Find the address of MGM studios.
- b) Find Sandra Bullock's birthdate.
- c) Find all the stars that appeared either in a movie made in 1980 or a movie with "Love" in the title.
- d) Find all executives worth at least \$10,000,000.
- e) Find all the stars who either are male or live in Malibu (have string **Malibu** as a part of their address).

**Exercise 6.1.3:** Write the following queries in SQL. They refer to the database schema of Exercise 2.4.1:

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

Show the result of your queries using the data from Exercise 2.4.1.

- a) Find the model number, speed, and hard-disk size for all PC's whose price is under \$1000.
- b) Do the same as (a), but rename the **speed** column **gigahertz** and the **hd** column **gigabytes**.
- c) Find the manufacturers of printers.
- d) Find the model number, memory size, and screen size for laptops costing more than \$1500.
- e) Find all the tuples in the **Printer** relation for color printers. Remember that **color** is a boolean-valued attribute.
- f) Find the model number and hard-disk size for those PC's that have a speed of 3.2 and a price less than \$2000.

**Exercise 6.1.4:** Write the following queries based on the database schema of Exercise 2.4.3:

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

and show the result of your query on the data of Exercise 2.4.3.

- a) Find the class name and country for all classes with at least 10 guns.
- b) Find the names of all ships launched prior to 1918, but call the resulting column **shipName**.
- c) Find the names of ships sunk in battle and the name of the battle in which they were sunk.
- d) Find all ships that have the same name as their class.
- e) Find the names of all ships that begin with the letter “R.”
- ! f) Find the names of all ships whose name consists of three or more words (e.g., King George V).

**Exercise 6.1.5:** Let  $a$  and  $b$  be integer-valued attributes that may be NULL in some tuples. For each of the following conditions (as may appear in a WHERE clause), describe exactly the set of  $(a, b)$  tuples that satisfy the condition, including the case where  $a$  and/or  $b$  is NULL.

- a)  $a = 10 \text{ OR } b = 20$
- b)  $a = 10 \text{ AND } b = 20$
- c)  $a < 10 \text{ OR } a \geq 10$
- ! d)  $a = b$
- ! e)  $a \leq b$

**! Exercise 6.1.6:** In Example 6.10 we discussed the query

```
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120;
```

which behaves unintuitively when the length of a movie is NULL. Find a simpler, equivalent query, one with a single condition in the WHERE clause (no AND or OR of conditions).

## 6.2 Queries Involving More Than One Relation

Much of the power of relational algebra comes from its ability to combine two or more relations through joins, products, unions, intersections, and differences. We get all of these operations in SQL. The set-theoretic operations — union, intersection, and difference — appear directly in SQL, as we shall learn in Section 6.2.5. First, we shall learn how the select-from-where statement of SQL allows us to perform products and joins.

### 6.2.1 Products and Joins in SQL

SQL has a simple way to couple relations in one query: list each relation in the **FROM** clause. Then, the **SELECT** and **WHERE** clauses can refer to the attributes of any of the relations in the **FROM** clause.

**Example 6.12:** Suppose we want to know the name of the producer of *Star Wars*. To answer this question we need the following two relations from our running example:

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

The producer certificate number is given in the **Movies** relation, so we can do a simple query on **Movies** to get this number. We could then do a second query on the relation **MovieExec** to find the name of the person with that certificate number.

However, we can phrase both these steps as one query about the pair of relations **Movies** and **MovieExec** as follows:

```
SELECT name
  FROM Movies, MovieExec
 WHERE title = 'Star Wars' AND producerC# = cert#;
```

This query asks us to consider all pairs of tuples, one from **Movies** and the other from **MovieExec**. The conditions on this pair are stated in the **WHERE** clause:

1. The **title** component of the tuple from **Movies** must have value '*Star Wars*'.
2. The **producerC#** attribute of the **Movies** tuple must be the same certificate number as the **cert#** attribute in the **MovieExec** tuple. That is, these two tuples must refer to the same producer.

Whenever we find a pair of tuples satisfying both conditions, we produce the **name** attribute of the tuple from **MovieExec** as part of the answer. If the data is what we expect, the only time both conditions will be met is when the tuple from **Movies** is for *Star Wars*, and the tuple from **MovieExec** is for George Lucas. Then and only then will the title be correct and the certificate numbers agree. Thus, **George Lucas** should be the only value produced. This process is suggested in Fig. 6.3. We take up in more detail how to interpret multirelation queries in Section 6.2.4. □

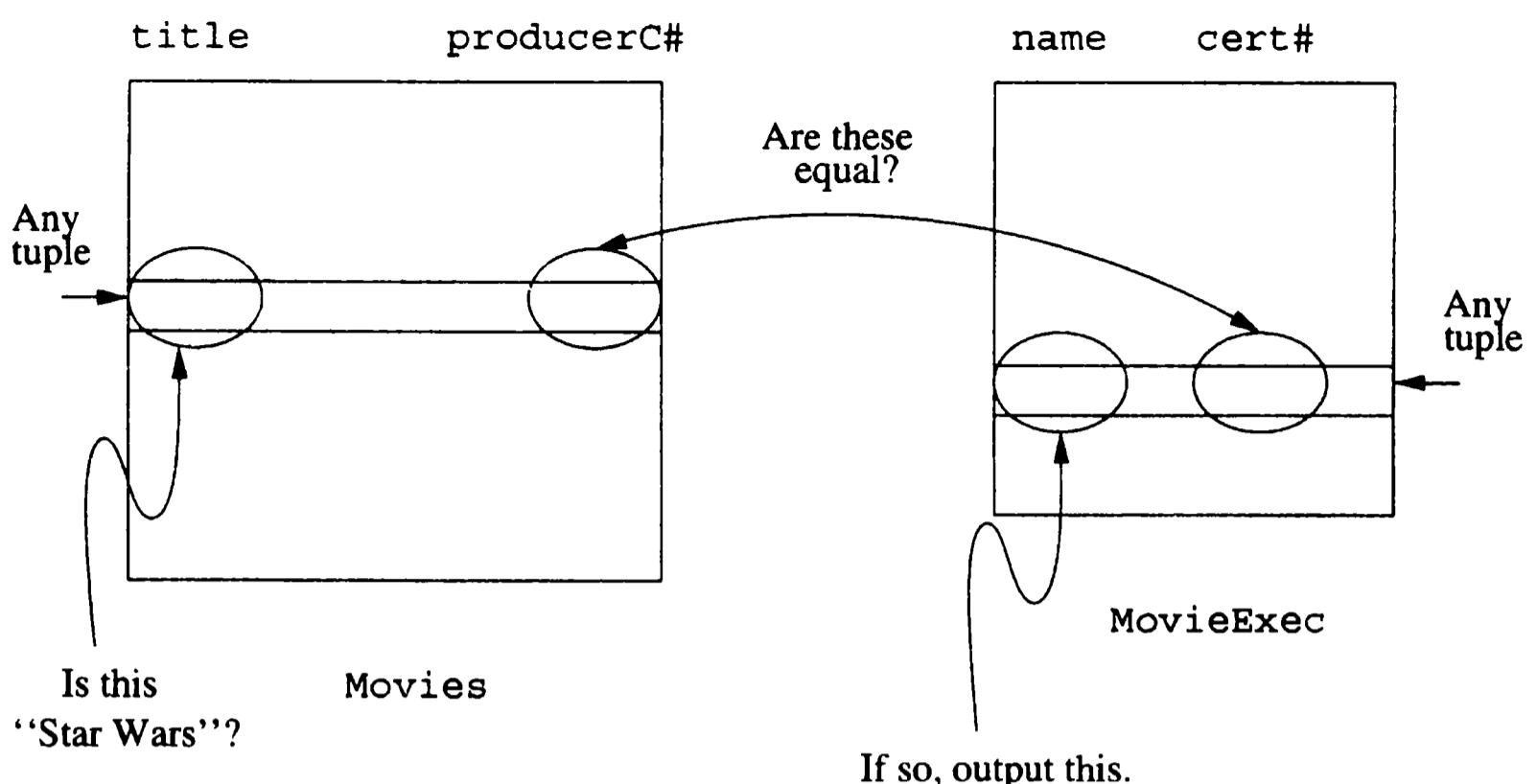


Figure 6.3: The query of Example 6.12 asks us to pair every tuple of **Movies** with every tuple of **MovieExec** and test two conditions

### 6.2.2 Disambiguating Attributes

Sometimes we ask a query involving several relations, and among these relations are two or more attributes with the same name. If so, we need a way to indicate which of these attributes is meant by a use of their shared name. SQL solves this problem by allowing us to place a relation name and a dot in front of an attribute. Thus  $R.A$  refers to the attribute  $A$  of relation  $R$ .

**Example 6.13:** The two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

each have attributes **name** and **address**. Suppose we wish to find pairs consisting of a star and an executive with the same address. The following query does the job.

```
SELECT MovieStar.name, MovieExec.name
FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

In this query, we look for a pair of tuples, one from **MovieStar** and the other from **MovieExec**, such that their address components agree. The **WHERE** clause enforces the requirement that the **address** attributes from each of the two tuples agree. Then, for each matching pair of tuples, we extract the two **name** attributes, first from the **MovieStar** tuple and then from the other. The result would be a set of pairs such as

<i>MovieStar.name</i>	<i>MovieExec.name</i>
Jane Fonda	Ted Turner
...	...

□

The relation name, followed by a dot, is permissible even in situations where there is no ambiguity. For instance, we are free to write the query of Example 6.12 as

```
SELECT MovieExec.name
  FROM Movies, MovieExec
 WHERE Movie.title = 'Star Wars'
   AND Movie.producerC# = MovieExec.cert#;
```

Alternatively, we may use relation names and dots in front of any subset of the attributes in this query.

### 6.2.3 Tuple Variables

Disambiguating attributes by prefixing the relation name works as long as the query involves combining several different relations. However, sometimes we need to ask a query that involves two or more tuples from the same relation. We may list a relation  $R$  as many times as we need to in the `FROM` clause, but we need a way to refer to each occurrence of  $R$ . SQL allows us to define, for each occurrence of  $R$  in the `FROM` clause, an “alias” which we shall refer to as a *tuple variable*. Each use of  $R$  in the `FROM` clause is followed by the (optional) keyword `AS` and the name of the tuple variable; we shall generally omit the `AS` in this context.

In the `SELECT` and `WHERE` clauses, we can disambiguate attributes of  $R$  by preceding them by the appropriate tuple variable and a dot. Thus, the tuple variable serves as another name for relation  $R$  and can be used in its place when we wish.

**Example 6.14:** While Example 6.13 asked for a star and an executive sharing an address, we might similarly want to know about two stars who share an address. The query is essentially the same, but now we must think of two tuples chosen from relation `MovieStar`, rather than tuples from each of `MovieStar` and `MovieExec`. Using tuple variables as aliases for two uses of `MovieStar`, we can write the query as

```
SELECT Star1.name, Star2.name
  FROM MovieStar Star1, MovieStar Star2
 WHERE Star1.address = Star2.address
   AND Star1.name < Star2.name;
```

## Tuple Variables and Relation Names

Technically, references to attributes in `SELECT` and `WHERE` clauses are *always* to a tuple variable. However, if a relation appears only once in the `FROM` clause, then we can use the relation name as its own tuple variable. Thus, we can see a relation name  $R$  in the `FROM` clause as shorthand for  $R \text{ AS } R$ . Furthermore, as we have seen, when an attribute belongs unambiguously to one relation, the relation name (tuple variable) may be omitted.

We see in the `FROM` clause the declaration of two tuple variables, `Star1` and `Star2`; each is an alias for relation `MovieStar`. The tuple variables are used in the `SELECT` clause to refer to the `name` components of the two tuples. These aliases are also used in the `WHERE` clause to say that the two `MovieStar` tuples represented by `Star1` and `Star2` have the same value in their `address` components.

The second condition in the `WHERE` clause, `Star1.name < Star2.name`, says that the name of the first star precedes the name of the second star alphabetically. If this condition were omitted, then tuple variables `Star1` and `Star2` could both refer to the same tuple. We would find that the two tuple variables referred to tuples whose `address` components are equal, of course, and thus produce each star name paired with itself.<sup>3</sup> The second condition also forces us to produce each pair of stars with a common address only once, in alphabetical order. If we used `<>` (not-equal) as the comparison operator, then we would produce pairs of married stars twice, like

<code>Star1.name</code>	<code>Star2.name</code>
Paul Newman	Joanne Woodward
Joanne Woodward	Paul Newman
...	...

□

### 6.2.4 Interpreting Multirelation Queries

There are several ways to define the meaning of the select-from-where expressions that we have just covered. All are *equivalent*, in the sense that they each give the same answer for each query applied to the same relation instances. We shall consider each in turn.

<sup>3</sup>A similar problem occurs in Example 6.13 when the same individual is both a star and an executive. We could solve that problem by requiring that the two names be unequal.

## Nested Loops

The semantics that we have implicitly used in examples so far is that of tuple variables. Recall that a tuple variable ranges over all tuples of the corresponding relation. A relation name that is not aliased is also a tuple variable ranging over the relation itself, as we mentioned in the box on “Tuple Variables and Relation Names.” If there are several tuple variables, we may imagine nested loops, one for each tuple variable, in which the variables each range over the tuples of their respective relations. For each assignment of tuples to the tuple variables, we decide whether the **WHERE** clause is true. If so, we produce a tuple consisting of the values of the expressions following **SELECT**; note that each term is given a value by the current assignment of tuples to tuple variables. This query-answering algorithm is suggested by Fig. 6.4.

```

LET the tuple variables in the from-clause range over
    relations  $R_1, R_2, \dots, R_n$ ;
FOR each tuple  $t_1$  in relation  $R_1$  DO
    FOR each tuple  $t_2$  in relation  $R_2$  DO
        ...
        FOR each tuple  $t_n$  in relation  $R_n$  DO
            IF the where-clause is satisfied when the values
            from  $t_1, t_2, \dots, t_n$  are substituted for all
            attribute references THEN
                evaluate the expressions of the select-clause
                according to  $t_1, t_2, \dots, t_n$  and produce the
                tuple of values that results.

```

Figure 6.4: Answering a simple SQL query

## Parallel Assignment

There is an equivalent definition in which we do not explicitly create nested loops ranging over the tuple variables. Rather, we consider in arbitrary order, or in parallel, all possible assignments of tuples from the appropriate relations to the tuple variables. For each such assignment, we consider whether the **WHERE** clause becomes true. Each assignment that produces a true **WHERE** clause contributes a tuple to the answer; that tuple is constructed from the attributes of the **SELECT** clause, evaluated according to that assignment.

## Conversion to Relational Algebra

A third approach is to relate the SQL query to relational algebra. We start with the tuple variables in the **FROM** clause and take the Cartesian product of their relations. If two tuple variables refer to the same relation, then this relation appears twice in the product, and we rename its attributes so all attributes have

## An Unintuitive Consequence of SQL Semantics

Suppose  $R$ ,  $S$ , and  $T$  are unary (one-component) relations, each having attribute  $A$  alone, and we wish to find those elements that are in  $R$  and also in either  $S$  or  $T$  (or both). That is, we want to compute  $R \cap (S \cup T)$ . We might expect the following SQL query would do the job.

```
SELECT R.A
FROM R, S, T
WHERE R.A = S.A OR R.A = T.A;
```

However, consider the situation in which  $T$  is empty. Since then  $R.A = T.A$  can never be satisfied, we might expect the query to produce exactly  $R \cap S$ , based on our intuition about how “OR” operates. Yet whichever of the three equivalent definitions of Section 6.2.4 one prefers, we find that the result is empty, regardless of how many elements  $R$  and  $S$  have in common. If we use the nested-loop semantics of Figure 6.4, then we see that the loop for tuple variable  $T$  iterates 0 times, since there are no tuples in the relation for the tuple variable to range over. Thus, the if-statement inside the for-loops never executes, and nothing can be produced. Similarly, if we look for assignments of tuples to the tuple variables, there is no way to assign a tuple to  $T$ , so no assignments exist. Finally, if we use the Cartesian-product approach, we start with  $R \times S \times T$ , which is empty because  $T$  is empty.

unique names. Similarly, attributes of the same name from different relations are renamed to avoid ambiguity.

Having created the product, we apply a selection operator to it by converting the **WHERE** clause to a selection condition in the obvious way. That is, each attribute reference in the **WHERE** clause is replaced by the attribute of the product to which it corresponds. Finally, we create from the **SELECT** clause a list of expressions for a final (extended) projection operation. As we did for the **WHERE** clause, we interpret each attribute reference in the **SELECT** clause as the corresponding attribute in the product of relations.

**Example 6.15 :** Let us convert the query of Example 6.14 to relational algebra. First, there are two tuple variables in the **FROM** clause, both referring to relation **MovieStar**. Thus, our expression (without the necessary renaming) begins:

**MovieStar**  $\times$  **MovieStar**

The resulting relation has eight attributes, the first four correspond to attributes **name**, **address**, **gender**, and **birthdate** from the first copy of relation **MovieStar**, and the second four correspond to the same attributes from the

other copy of `MovieStar`. We could create names for these attributes with a dot and the aliasing tuple variable — e.g., `Star1.gender` — but for succinctness, let us invent new symbols and call the attributes simply  $A_1, A_2, \dots, A_8$ . Thus,  $A_1$  corresponds to `Star1.name`,  $A_5$  corresponds to `Star2.name`, and so on.

Under this naming strategy for attributes, the selection condition obtained from the `WHERE` clause is  $A_2 = A_6$  and  $A_1 < A_5$ . The projection list is  $A_1, A_5$ . Thus,

$$\pi_{A_1, A_5} \left( \sigma_{A_2 = A_6 \text{ AND } A_1 < A_5} \left( \rho_{M(A_1, A_2, A_3, A_4)}(\text{MovieStar}) \times \rho_{N(A_5, A_6, A_7, A_8)}(\text{MovieStar}) \right) \right)$$

renders the entire query in relational algebra.  $\square$

### 6.2.5 Union, Intersection, and Difference of Queries

Sometimes we wish to combine relations using the set operations of relational algebra: union, intersection, and difference. SQL provides corresponding operators that apply to the results of queries, provided those queries produce relations with the same list of attributes and attribute types. The keywords used are `UNION`, `INTERSECT`, and `EXCEPT` for  $\cup$ ,  $\cap$ , and  $-$ , respectively. Words like `UNION` are used between two queries, and those queries must be parenthesized.

**Example 6.16:** Suppose we wanted the names and addresses of all female movie stars who are also movie executives with a net worth over \$10,000,000. Using the following two relations:

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

we can write the query as in Fig. 6.5. Lines (1) through (3) produce a relation whose schema is `(name, address)` and whose tuples are the names and addresses of all female movie stars.

```
1)  (SELECT name, address
2)    FROM MovieStar
3)    WHERE gender = 'F')
4)      INTERSECT
5)  (SELECT name, address
6)    FROM MovieExec
7)    WHERE netWorth > 10000000);
```

Figure 6.5: Intersecting female movie stars with rich executives

Similarly, lines (5) through (7) produce the set of “rich” executives, those with net worth over \$10,000,000. This query also yields a relation whose schema

## Readable SQL Queries

Generally, one writes SQL queries so that each important keyword like `FROM` or `WHERE` starts a new line. This style offers the reader visual clues to the structure of the query. However, when a query or subquery is short, we shall sometimes write it out on a single line, as we did in Example 6.17. That style, keeping a complete query compact, also offers good readability.

has the attributes `name` and `address` only. Since the two schemas are the same, we can intersect them, and we do so with the operator of line (4). □

**Example 6.17:** In a similar vein, we could take the difference of two sets of persons, each selected from a relation. The query

```
(SELECT name, address FROM MovieStar)
EXCEPT
(SELECT name, address FROM MovieExec);
```

gives the names and addresses of movie stars who are not also movie executives, regardless of gender or net worth. □

In the two examples above, the attributes of the relations whose intersection or difference we took were conveniently the same. However, if necessary to get a common set of attributes, we can rename attributes as in Example 6.3.

**Example 6.18:** Suppose we wanted all the titles and years of movies that appeared in either the `Movies` or `StarsIn` relation of our running example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

Ideally, these sets of movies would be the same, but in practice it is common for relations to diverge; for instance we might have movies with no listed stars or a `StarsIn` tuple that mentions a movie not found in the `Movies` relation.<sup>4</sup> Thus, we might write

```
(SELECT title, year FROM Movie)
UNION
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

The result would be all movies mentioned in either relation, with `title` and `year` as the attributes of the resulting relation. □

---

<sup>4</sup>There are ways to prevent this divergence; see Section 7.1.1.

### 6.2.6 Exercises for Section 6.2

**Exercise 6.2.1:** Using the database schema of our running movie example

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

write the following queries in SQL.

- a) Who were the male stars in *Titanic*?
- b) Which stars appeared in movies produced by MGM in 1995?
- c) Who is the president of MGM studios?
- ! d) Which movies are longer than *Gone With the Wind*?
- ! e) Which executives are worth more than Merv Griffin?

**Exercise 6.2.2:** Write the following queries, based on the database schema

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

of Exercise 2.4.1, and evaluate your queries using the data of that exercise.

- a) Give the manufacturer and speed of laptops with a hard disk of at least thirty gigabytes.
- b) Find the model number and price of all products (of any type) made by manufacturer *B*.
- c) Find those manufacturers that sell Laptops, but not PC's.
- ! d) Find those hard-disk sizes that occur in two or more PC's.
- ! e) 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  $(j, i)$ .
- !! f) Find those manufacturers of at least two different computers (PC's or laptops) with speeds of at least 3.0.

**Exercise 6.2.3:** Write the following queries, based on the database schema

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

of Exercise 2.4.3, and evaluate your queries using the data of that exercise.

- a) Find the ships heavier than 35,000 tons.
- b) List the name, displacement, and number of guns of the ships engaged in the battle of Guadalcanal.
- c) List all the ships mentioned in the database. (Remember that all these ships may not appear in the `Ships` relation.)
- ! d) Find those countries that have both battleships and battlecruisers.
- ! e) Find those ships that were damaged in one battle, but later fought in another.
- ! f) Find those battles with at least three ships of the same country.

**! Exercise 6.2.4:** A general form of relational-algebra query is

$$\pi_L \left( \sigma_C (R_1 \times R_2 \times \dots \times R_n) \right)$$

Here,  $L$  is an arbitrary list of attributes, and  $C$  is an arbitrary condition. The list of relations  $R_1, R_2, \dots, R_n$  may include the same relation repeated several times, in which case appropriate renaming may be assumed applied to the  $R_i$ 's. Show how to express any query of this form in SQL.

**! Exercise 6.2.5:** Another general form of relational-algebra query is

$$\pi_L \left( \sigma_C (R_1 \bowtie R_2 \bowtie \dots \bowtie R_n) \right)$$

The same assumptions as in Exercise 6.2.4 apply here; the only difference is that the natural join is used instead of the product. Show how to express any query of this form in SQL.

## 6.3 Subqueries

In SQL, one query can be used in various ways to help in the evaluation of another. A query that is part of another is called a *subquery*. Subqueries can have subqueries, and so on, down as many levels as we wish. We already saw one example of the use of subqueries; in Section 6.2.5 we built a union, intersection, or difference query by connecting two subqueries to form the whole query. There are a number of other ways that subqueries can be used:

1. Subqueries can return a single constant, and this constant can be compared with another value in a **WHERE** clause.
2. Subqueries can return relations that can be used in various ways in **WHERE** clauses.
3. Subqueries can appear in **FROM** clauses, followed by a tuple variable that represents the tuples in the result of the subquery.

### 6.3.1 Subqueries that Produce Scalar Values

An atomic value that can appear as one component of a tuple is referred to as a *scalar*. A select-from-where expression can produce a relation with any number of attributes in its schema, and there can be any number of tuples in the relation. However, often we are only interested in values of a single attribute. Furthermore, sometimes we can deduce from information about keys, or from other information, that there will be only a single value produced for that attribute.

If so, we can use this select-from-where expression, surrounded by parentheses, as if it were a constant. In particular, it may appear in a **WHERE** clause any place we would expect to find a constant or an attribute representing a component of a tuple. For instance, we may compare the result of such a subquery to a constant or attribute.

**Example 6.19:** Let us recall Example 6.12, where we asked for the producer of *Star Wars*. We had to query the two relations

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

because only the former has movie title information and only the latter has producer names. The information is linked by “certificate numbers.” These numbers uniquely identify producers. The query we developed is:

```
SELECT name
  FROM Movies, MovieExec
 WHERE title = 'Star Wars' AND producerC# = cert#;
```

There is another way to look at this query. We need the **Movies** relation only to get the certificate number for the producer of *Star Wars*. Once we have it, we can query the relation **MovieExec** to find the name of the person with this certificate. The first problem, getting the certificate number, can be written as a subquery, and the result, which we expect will be a single value, can be used in the “main” query to achieve the same effect as the query above. This query is shown in Fig. 6.6.

Lines (4) through (6) of Fig. 6.6 are the subquery. Looking only at this simple query by itself, we see that the result will be a unary relation with

```

1) SELECT name
2) FROM MovieExec
3) WHERE cert# =
4)     (SELECT producerC#
5)         FROM Movies
6)         WHERE title = 'Star Wars'
);

```

Figure 6.6: Finding the producer of *Star Wars* by using a nested subquery

attribute `producerC#`, and we expect to find only one tuple in this relation. The tuple will look like (12345), that is, a single component with some integer, perhaps 12345 or whatever George Lucas' certificate number is. If zero tuples or more than one tuple is produced by the subquery of lines (4) through (6), it is a run-time error.

Having executed this subquery, we can then execute lines (1) through (3) of Fig. 6.6, as if the value 12345 replaced the entire subquery. That is, the “main” query is executed as if it were

```

SELECT name
FROM MovieExec
WHERE cert# = 12345;

```

The result of this query should be **George Lucas**. □

### 6.3.2 Conditions Involving Relations

There are a number of SQL operators that we can apply to a relation  $R$  and produce a boolean result. However, the relation  $R$  must be expressed as a subquery. As a trick, if we want to apply these operators to a stored table `Foo`, we can use the subquery (`SELECT * FROM Foo`). The same trick works for union, intersection, and difference of relations. Notice that those operators, introduced in Section 6.2.5 are applied to two subqueries.

Some of the operators below — `IN`, `ALL`, and `ANY` — will be explained first in their simple form where a scalar value  $s$  is involved. In this situation, the subquery  $R$  is required to produce a one-column relation. Here are the definitions of the operators:

1. `EXISTS R` is a condition that is true if and only if  $R$  is not empty.
2.  $s \text{ IN } R$  is true if and only if  $s$  is equal to one of the values in  $R$ . Likewise,  $s \text{ NOT IN } R$  is true if and only if  $s$  is equal to no value in  $R$ . Here, we assume  $R$  is a unary relation. We shall discuss extensions to the `IN` and `NOT IN` operators where  $R$  has more than one attribute in its schema and  $s$  is a tuple in Section 6.3.3.

3.  $s > \text{ALL } R$  is true if and only if  $s$  is greater than every value in unary relation  $R$ . Similarly, the  $>$  operator could be replaced by any of the other five comparison operators, with the analogous meaning:  $s$  stands in the stated relationship to every tuple in  $R$ . For instance,  $s <> \text{ALL } R$  is the same as  $s \text{ NOT IN } R$ .
4.  $s > \text{ANY } R$  is true if and only if  $s$  is greater than at least one value in unary relation  $R$ . Similarly, any of the other five comparisons could be used in place of  $>$ , with the meaning that  $s$  stands in the stated relationship to at least one tuple of  $R$ . For instance,  $s = \text{ANY } R$  is the same as  $s \text{ IN } R$ .

The **EXISTS**, **ALL**, and **ANY** operators can be negated by putting **NOT** in front of the entire expression, just like any other boolean-valued expression. Thus, **NOT EXISTS**  $R$  is true if and only if  $R$  is empty. **NOT**  $s \geq \text{ALL } R$  is true if and only if  $s$  is not the maximum value in  $R$ , and **NOT**  $s > \text{ANY } R$  is true if and only if  $s$  is the minimum value in  $R$ . We shall see several examples of the use of these operators shortly.

### 6.3.3 Conditions Involving Tuples

A tuple in SQL is represented by a parenthesized list of scalar values. Examples are `(123, 'foo')` and `(name, address, networth)`. The first of these has constants as components; the second has attributes as components. Mixing of constants and attributes is permitted.

If a tuple  $t$  has the same number of components as a relation  $R$ , then it makes sense to compare  $t$  and  $R$  in expressions of the type listed in Section 6.3.2. Examples are  $t \text{ IN } R$  or  $t <> \text{ANY } R$ . The latter comparison means that there is some tuple in  $R$  other than  $t$ . Note that when comparing a tuple with members of a relation  $R$ , we must compare components using the assumed standard order for the attributes of  $R$ .

```

1)  SELECT name
2)  FROM MovieExec
3)  WHERE cert# IN
4)      (SELECT producerC#
5)          FROM Movies
6)          WHERE (title, year) IN
7)              (SELECT movieTitle, movieYear
8)                  FROM StarsIn
9)                  WHERE starName = 'Harrison Ford'
)
);

```

Figure 6.7: Finding the producers of Harrison Ford's movies

**Example 6.20:** In Fig. 6.7 is a SQL query on the three relations

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

asking for all the producers of movies in which Harrison Ford stars. It consists of a “main” query, a query nested within that, and a third query nested within the second.

We should analyze any query with subqueries from the inside out. Thus, let us start with the innermost nested subquery: lines (7) through (9). This query examines the tuples of the relation **StarsIn** and finds all those tuples whose **starName** component is ‘Harrison Ford’. The titles and years of those movies are returned by this subquery. Recall that title and year, not title alone, is the key for movies, so we need to produce tuples with both attributes to identify a movie uniquely. Thus, we would expect the value produced by lines (7) through (9) to look something like Fig. 6.8.

<i>title</i>	<i>year</i>
Star Wars	1977
Raiders of the Lost Ark	1981
The Fugitive	1993
...	...

Figure 6.8: Title-year pairs returned by inner subquery

Now, consider the middle subquery, lines (4) through (6). It searches the **Movies** relation for tuples whose title and year are in the relation suggested by Fig. 6.8. For each tuple found, the producer’s certificate number is returned, so the result of the middle subquery is the set of certificates of the producers of Harrison Ford’s movies.

Finally, consider the “main” query of lines (1) through (3). It examines the tuples of the **MovieExec** relation to find those whose **cert#** component is one of the certificates in the set returned by the middle subquery. For each of these tuples, the name of the producer is returned, giving us the set of producers of Harrison Ford’s movies, as desired. □

Incidentally, the nested query of Fig. 6.7 can, like many nested queries, be written as a single select-from-where expression with relations in the **FROM** clause for each of the relations mentioned in the main query or a subquery. The **IN** relationships are replaced by equalities in the **WHERE** clause. For instance, the query of Fig. 6.9 is essentially that of Fig. 6.7. There is a difference regarding the way duplicate occurrences of a producer — e.g., George Lucas — are handled, as we shall discuss in Section 6.4.1.

```

SELECT name
FROM MovieExec, Movies, StarsIn
WHERE cert# = producerC# AND
      title = movieTitle AND
      year = movieYear AND
      starName = 'Harrison Ford';

```

Figure 6.9: Ford's producers without nested subqueries

### 6.3.4 Correlated Subqueries

The simplest subqueries can be evaluated once and for all, and the result used in a higher-level query. A more complicated use of nested subqueries requires the subquery to be evaluated many times, once for each assignment of a value to some term in the subquery that comes from a tuple variable outside the subquery. A subquery of this type is called a *correlated* subquery. Let us begin our study with an example.

**Example 6.21:** We shall find the titles that have been used for two or more movies. We start with an outer query that looks at all tuples in the relation

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

For each such tuple, we ask in a subquery whether there is a movie with the same title and a greater year. The entire query is shown in Fig. 6.10.

As with other nested queries, let us begin at the innermost subquery, lines (4) through (6). If `Old.title` in line (6) were replaced by a constant string such as '`King Kong`', we would understand it quite easily as a query asking for the year or years in which movies titled *King Kong* were made. The present subquery differs little. The only problem is that we don't know what value `Old.title` has. However, as we range over `Movies` tuples of the outer query of lines (1) through (3), each tuple provides a value of `Old.title`. We then execute the query of lines (4) through (6) with this value for `Old.title` to decide the truth of the `WHERE` clause that extends from lines (3) through (6).

```

1) SELECT title
2) FROM Movies Old
3) WHERE year < ANY
4)     (SELECT year
5)         FROM Movies
6)         WHERE title = Old.title
);

```

Figure 6.10: Finding movie titles that appear more than once

The condition of line (3) is true if any movie with the same title as `Old.title` has a later year than the movie in the tuple that is the current value of tuple variable `Old`. This condition is true unless the year in the tuple `Old` is the last year in which a movie of that title was made. Consequently, lines (1) through (3) produce a title one fewer times than there are movies with that title. A movie made twice will be listed once, a movie made three times will be listed twice, and so on.<sup>5</sup> □

When writing a correlated query it is important that we be aware of the *scoping rules* for names. In general, an attribute in a subquery belongs to one of the tuple variables in that subquery's `FROM` clause if some tuple variable's relation has that attribute in its schema. If not, we look at the immediately surrounding subquery, then to the one surrounding that, and so on. Thus, `year` on line (4) and `title` on line (6) of Fig. 6.10 refer to the attributes of the tuple variable that ranges over all the tuples of the copy of relation `Movies` introduced on line (5) — that is, the copy of the `Movies` relation addressed by the subquery of lines (4) through (6).

However, we can arrange for an attribute to belong to another tuple variable if we prefix it by that tuple variable and a dot. That is why we introduced the alias `Old` for the `Movies` relation of the outer query, and why we refer to `Old.title` in line (6). Note that if the two relations in the `FROM` clauses of lines (2) and (5) were different, we would not need an alias. Rather, in the subquery we could refer directly to attributes of a relation mentioned in line (2).

### 6.3.5 Subqueries in `FROM` Clauses

Another use for subqueries is as relations in a `FROM` clause. In a `FROM` list, instead of a stored relation, we may use a parenthesized subquery. Since we don't have a name for the result of this subquery, we must give it a tuple-variable alias. We then refer to tuples in the result of the subquery as we would tuples in any relation that appears in the `FROM` list.

**Example 6.22:** Let us reconsider the problem of Example 6.20, where we wrote a query that finds the producers of Harrison Ford's movies. Suppose we had a relation that gave the certificates of the producers of those movies. It would then be a simple matter to look up the names of those producers in the relation `MovieExec`. Figure 6.11 is such a query.

Lines (2) through (7) are the `FROM` clause of the outer query. In addition to the relation `MovieExec`, it has a subquery. That subquery joins `Movies` and `StarsIn` on lines (3) through (5), adds the condition that the star is Harrison Ford on line (6), and returns the set of producers of the movies at line (2). This set is given the alias `Prod` on line (7).

---

<sup>5</sup>This example is the first occasion on which we've been reminded that relations in SQL are bags, not sets. There are several ways that duplicates may crop up in SQL relations. We shall discuss the matter in detail in Section 6.4.

```

1)  SELECT name
2)  FROM MovieExec, (SELECT producerC#
3)                                FROM Movies, StarsIn
4)                                WHERE title = movieTitle AND
5)                                year = movieYear AND
6)                                starName = 'Harrison Ford'
7)                            ) Prod
8)  WHERE cert# = Prod.producerC#;

```

Figure 6.11: Finding the producers of Ford’s movies using a subquery in the `FROM` clause

At line (8), the relations `MovieExec` and the subquery aliased `Prod` are joined with the requirement that the certificate numbers be the same. The names of the producers from `MovieExec` that have certificates in the set aliased by `Prod` is returned at line (1). □

### 6.3.6 SQL Join Expressions

We can construct relations by a number of variations on the join operator applied to two relations. These variants include products, natural joins, theta-joins, and outerjoins. The result can stand as a query by itself. Alternatively, all these expressions, since they produce relations, may be used as subqueries in the `FROM` clause of a select-from-where expression. These expressions are principally shorthands for more complex select-from-where queries (see Exercise 6.3.11).

The simplest form of join expression is a *cross join*; that term is a synonym for what we called a Cartesian product or just “product” in Section 2.4.7. For instance, if we want the product of the two relations

```

Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)

```

we can say

```
Movies CROSS JOIN StarsIn;
```

and the result will be a nine-column relation with all the attributes of `Movies` and `StarsIn`. Every pair consisting of one tuple of `Movies` and one tuple of `StarsIn` will be a tuple of the resulting relation.

The attributes in the product relation can be called *R.A*, where *R* is one of the two joined relations and *A* is one of its attributes. If only one of the relations has an attribute named *A*, then the *R* and dot can be dropped, as usual. In this instance, since `Movies` and `StarsIn` have no common attributes, the nine attribute names suffice in the product.

However, the product by itself is rarely a useful operation. A more conventional theta-join is obtained with the keyword `ON`. We put `JOIN` between two

relation names  $R$  and  $S$  and follow them by `ON` and a condition. The meaning of `JOIN...ON` is that the product of  $R \times S$  is followed by a selection for whatever condition follows `ON`.

**Example 6.23:** Suppose we want to join the relations

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
```

with the condition that the only tuples to be joined are those that refer to the same movie. That is, the titles and years from both relations must be the same. We can ask this query by

```
Movies JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

The result is again a nine-column relation with the obvious attribute names. However, now a tuple from `Movies` and one from `StarsIn` combine to form a tuple of the result only if the two tuples agree on both the title and year. As a result, two of the columns are redundant, because every tuple of the result will have the same value in both the `title` and `movieTitle` components and will have the same value in both `year` and `movieYear`.

If we are concerned with the fact that the join above has two redundant components, we can use the whole expression as a subquery in a `FROM` clause and use a `SELECT` clause to remove the undesired attributes. Thus, we could write

```
SELECT title, year, length, genre, studioName,
      producerC#, starName
  FROM Movies JOIN StarsIn ON
        title = movieTitle AND year = movieYear;
```

to get a seven-column relation which is the `Movies` relation's tuples, each extended in all possible ways with a star of that movie.  $\square$

### 6.3.7 Natural Joins

As we recall from Section 2.4.8, a natural join differs from a theta-join in that:

1. The join condition is that all pairs of attributes from the two relations having a common name are equated, and there are no other conditions.
2. One of each pair of equated attributes is projected out.

The SQL natural join behaves exactly this way. Keywords `NATURAL JOIN` appear between the relations to express the  $\bowtie$  operator.

**Example 6.24:** Suppose we want to compute the natural join of the relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

The result will be a relation whose schema includes attributes `name` and `address` plus all the attributes that appear in one or the other of the two relations. A tuple of the result will represent an individual who is both a star and an executive and will have all the information pertinent to either: a name, address, gender, birthdate, certificate number, and net worth. The expression

```
MovieStar NATURAL JOIN MovieExec;
```

succinctly describes the desired relation.  $\square$

### 6.3.8 Outerjoins

The outerjoin operator was introduced in Section 5.2.7 as a way to augment the result of a join by the dangling tuples, padded with null values. In SQL, we can specify an outerjoin; `NULL` is used as the null value.

**Example 6.25:** Suppose we wish to take the outerjoin of the two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

SQL refers to the standard outerjoin, which pads dangling tuples from both of its arguments, as a *full* outerjoin. The syntax is unsurprising:

```
MovieStar NATURAL FULL OUTER JOIN MovieExec;
```

The result of this operation is a relation with the same six-attribute schema as Example 6.24. The tuples of this relation are of three kinds. Those representing individuals who are both stars and executives have tuples with all six attributes non-`NULL`. These are the tuples that are also in the result of Example 6.24.

The second kind of tuple is one for an individual who is a star but not an executive. These tuples have values for attributes `name`, `address`, `gender`, and `birthdate` taken from their tuple in `MovieStar`, while the attributes belonging only to `MovieExec`, namely `cert#` and `netWorth`, have `NULL` values.

The third kind of tuple is for an executive who is not also a star. These tuples have values for the attributes of `MovieExec` taken from their `MovieExec` tuple and `NULL`'s in the attributes `gender` and `birthdate` that come only from `MovieStar`. For instance, the three tuples of the result relation shown in Fig. 6.12 correspond to the three types of individuals, respectively.  $\square$

All the variations on the outerjoin that we mentioned in Section 5.2.7 are also available in SQL. If we want a left- or right-outerjoin, we add the appropriate word `LEFT` or `RIGHT` in place of `FULL`. For instance,

```
MovieStar NATURAL LEFT OUTER JOIN MovieExec;
```

<i>name</i>	<i>address</i>	<i>gender</i>	<i>birthdate</i>	<i>cert#</i>	<i>networth</i>
Mary Tyler Moore	Maple St.	'F'	9/9/99	12345	\$100...
Tom Hanks	Cherry Ln.	'M'	8/8/88	NULL	NULL
George Lucas	Oak Rd.	NULL	NULL	23456	\$200...

Figure 6.12: Three tuples in the outerjoin of **MovieStar** and **MovieExec**

would yield the first two tuples of Fig. 6.12 but not the third. Similarly,

```
MovieStar NATURAL RIGHT OUTER JOIN MovieExec;
```

would yield the first and third tuples of Fig. 6.12 but not the second.

Next, suppose we want a theta-outerjoin instead of a natural outerjoin. Instead of using the keyword **NATURAL**, we may follow the join by **ON** and a condition that matching tuples must obey. If we also specify **FULL OUTER JOIN**, then after matching tuples from the two joined relations, we pad dangling tuples of either relation with **NULL**'s and include the padded tuples in the result.

**Example 6.26 :** Let us reconsider Example 6.23, where we joined the relations **Movies** and **StarsIn** using the conditions that the **title** and **movieTitle** attributes of the two relations agree and that the **year** and **movieYear** attributes of the two relations agree. If we modify that example to call for a full outerjoin:

```
Movies FULL OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

then we shall get not only tuples for movies that have at least one star mentioned in **StarsIn**, but we shall get tuples for movies with no listed stars, padded with **NULL**'s in attributes **movieTitle**, **movieYear**, and **starName**. Likewise, for stars not appearing in any movie listed in relation **Movies** we get a tuple with **NULL**'s in the six attributes of **Movies**. □

The keyword **FULL** can be replaced by either **LEFT** or **RIGHT** in outerjoins of the type suggested by Example 6.26. For instance,

```
Movies LEFT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

gives us the **Movies** tuples with at least one listed star and **NULL**-padded **Movies** tuples without a listed star, but will not include stars without a listed movie. Conversely,

```
Movies RIGHT OUTER JOIN StarsIn ON
    title = movieTitle AND year = movieYear;
```

will omit the tuples for movies without a listed star but will include tuples for stars not in any listed movies, padded with **NULL**'s.

### 6.3.9 Exercises for Section 6.3

**Exercise 6.3.1:** Write the following queries, based on the database schema

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

of Exercise 2.4.1. You should use at least one subquery in each of your answers and write each query in two significantly different ways (e.g., using different sets of the operators EXISTS, IN, ALL, and ANY).

- a) Find the makers of PC's with a speed of at least 3.0.
- b) Find the printers with the highest price.
- c) Find the laptops whose speed is slower than that of any PC.
- d) Find the model number of the item (PC, laptop, or printer) with the highest price.
- e) Find the maker of the color printer with the lowest price.
- f) Find the maker(s) of the PC(s) with the fastest processor among all those PC's that have the smallest amount of RAM.

**Exercise 6.3.2:** Write the following queries, based on the database schema

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

of Exercise 2.4.3. You should use at least one subquery in each of your answers and write each query in two significantly different ways (e.g., using different sets of the operators EXISTS, IN, ALL, and ANY).

- a) Find the countries whose ships had the largest number of guns.
  - b) Find the classes of ships, at least one of which was sunk in a battle.
  - c) Find the names of the ships with a 16-inch bore.
  - d) Find the battles in which ships of the Kongo class participated.
  - e) Find the names of the ships whose number of guns was the largest for those ships of the same bore.
- SQL response 6.3.3:** Write the query of Fig. 6.10 without any subqueries.

- ! Exercise 6.3.4:** Consider expression  $\pi_L(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n)$  of relational algebra, where  $L$  is a list of attributes all of which belong to  $R_1$ . Show that this expression can be written in SQL using subqueries only. More precisely, write an equivalent SQL expression where no FROM clause has more than one relation in its list.
- ! Exercise 6.3.5:** Write the following queries without using the intersection or difference operators:
- The intersection query of Fig. 6.5.
  - The difference query of Example 6.17.
- !! Exercise 6.3.6:** We have noticed that certain operators of SQL are redundant, in the sense that they always can be replaced by other operators. For example, we saw that  $s \text{ IN } R$  can be replaced by  $s = \text{ANY } R$ . Show that EXISTS and NOT EXISTS are redundant by explaining how to replace any expression of the form EXISTS  $R$  or NOT EXISTS  $R$  by an expression that does not involve EXISTS (except perhaps in the expression  $R$  itself). *Hint:* Remember that it is permissible to have a constant in the SELECT clause.

**Exercise 6.3.7:** For these relations from our running movie database schema

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

describe the tuples that would appear in the following SQL expressions:

- Studio CROSS JOIN MovieExec;
- StarsIn NATURAL FULL OUTER JOIN MovieStar;
- StarsIn FULL OUTER JOIN MovieStar ON name = starName;

**! Exercise 6.3.8:** Using the database schema

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

write a SQL query that will produce information about all products — PC's, laptops, and printers — including their manufacturer if available, and whatever information about that product is relevant (i.e., found in the relation for that type of product).

**Exercise 6.3.9:** Using the two relations

- de tuples for

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

from our database schema of Exercise 2.4.3, write a SQL query that will produce all available information about ships, including that information available in the `Classes` relation. You need not produce information about classes if there are no ships of that class mentioned in `Ships`.

**! Exercise 6.3.10:** Repeat Exercise 6.3.9, but also include in the result, for any class  $C$  that is not mentioned in `Ships`, information about the ship that has the same name  $C$  as its class. You may assume that there is a ship with the class name, even if it doesn't appear in `Ships`.

**! Exercise 6.3.11:** The join operators (other than `outerjoin`) we learned in this section are redundant, in the sense that they can always be replaced by `select-from-where` expressions. Explain how to write expressions of the following forms using `select-from-where`:

- a) `R CROSS JOIN S;`
- b) `R NATURAL JOIN S;`
- c) `R JOIN S ON  $C$ ;`, where  $C$  is a SQL condition.

## 6.4 Full-Relation Operations

In this section we shall study some operations that act on relations as a whole, rather than on tuples individually or in small numbers (as do joins of several relations, for instance). First, we deal with the fact that SQL uses relations that are bags rather than sets, and a tuple can appear more than once in a relation. We shall see how to force the result of an operation to be a set in Section 6.4.1, and in Section 6.4.2 we shall see that it is also possible to prevent the elimination of duplicates in circumstances where SQL systems would normally eliminate them.

Then, we discuss how SQL supports the grouping and aggregation operator  $\gamma$  that we introduced in Section 5.2.4. SQL has aggregation operators and a `GROUP-BY` clause. There is also a “HAVING” clause that allows selection of certain groups in a way that depends on the group as a whole, rather than on individual tuples.

### 6.4.1 Eliminating Duplicates

As mentioned in Section 6.3.4, SQL's notion of relations differs from the abstract notion of relations presented in Section 2.2. A relation, being a set, cannot have more than one copy of any given tuple. When a SQL query creates a new relation, the SQL system does not ordinarily eliminate duplicates. Thus, the SQL response to a query may list the same tuple several times.

Recall from Section 6.2.4 that one of several equivalent definitions of the meaning of a SQL select-from-where query is that we begin with the Cartesian product of the relations referred to in the **FROM** clause. Each tuple of the product is tested by the condition in the **WHERE** clause, and the ones that pass the test are given to the output for projection according to the **SELECT** clause. This projection may cause the same tuple to result from different tuples of the product, and if so, each copy of the resulting tuple is printed in its turn. Further, since there is nothing wrong with a SQL relation having duplicates, the relations from which the Cartesian product is formed may have duplicates, and each identical copy is paired with the tuples from the other relations, yielding a proliferation of duplicates in the product.

If we do not wish duplicates in the result, then we may follow the keyword **SELECT** by the keyword **DISTINCT**. That word tells SQL to produce only one copy of any tuple and is the SQL analog of applying the  $\delta$  operator of Section 5.2.1 to the result of the query.

**Example 6.27:** Let us reconsider the query of Fig. 6.9, where we asked for the producers of Harrison Ford’s movies using no subqueries. As written, George Lucas will appear many times in the output. If we want only to see each producer once, we may change line (1) of the query to

1) **SELECT DISTINCT name**

Then, the list of producers will have duplicate occurrences of names eliminated before printing.

Incidentally, the query of Fig. 6.7, where we used subqueries, does not necessarily suffer from the problem of duplicate answers. True, the subquery at line (4) of Fig. 6.7 will produce the certificate number of George Lucas several times. However, in the “main” query of line (1), we examine each tuple of **MovieExec** once. Presumably, there is only one tuple for George Lucas in that relation, and if so, it is only this tuple that satisfies the **WHERE** clause of line (3). Thus, George Lucas is printed only once.  $\square$

## 6.4.2 Duplicates in Unions, Intersections, and Differences

Unlike the **SELECT** statement, which preserves duplicates as a default and only eliminates them when instructed to by the **DISTINCT** keyword, the union, intersection, and difference operations, which we introduced in Section 6.2.5, normally eliminate duplicates. That is, bags are converted to sets, and the set version of the operation is applied. In order to prevent the elimination of duplicates, we must follow the operator **UNION**, **INTERSECT**, or **EXCEPT** by the keyword **ALL**. If we do, then we get the bag semantics of these operators as was discussed in Section 5.1.2.

**Example 6.28:** Consider again the union expression from Example 6.18, but now add the keyword **ALL**, as:

## The Cost of Duplicate Elimination

One might be tempted to place DISTINCT after every SELECT, on the theory that it is harmless. In fact, it is very expensive to eliminate duplicates from a relation. The relation must be sorted or partitioned so that identical tuples appear next to each other. Only by grouping the tuples in this way can we determine whether or not a given tuple should be eliminated. The time it takes to sort the relation so that duplicates may be eliminated is often greater than the time it takes to execute the query itself. Thus, duplicate elimination should be used judiciously if we want our queries to run fast.

```
(SELECT title, year FROM Movies)
    UNION ALL
(SELECT movieTitle AS title, movieYear AS year FROM StarsIn);
```

Now, a title and year will appear as many times in the result as it appears in each of the relations **Movies** and **StarsIn** put together. For instance, if a movie appeared once in the **Movies** relation and there were three stars for that movie listed in **StarsIn** (so the movie appeared in three different tuples of **StarsIn**), then that movie's title and year would appear four times in the result of the union. □

As for union, the operators **INTERSECT ALL** and **EXCEPT ALL** are intersection and difference of bags. Thus, if  $R$  and  $S$  are relations, then the result of expression

$$R \text{ INTERSECT ALL } S$$

is the relation in which the number of times a tuple  $t$  appears is the minimum of the number of times it appears in  $R$  and the number of times it appears in  $S$ .

The result of expression

$$R \text{ EXCEPT ALL } S$$

has tuple  $t$  as many times as the difference of the number of times it appears in  $R$  minus the number of times it appears in  $S$ , provided the difference is positive. Each of these definitions is what we discussed for bags in Section 5.1.2.

### 6.4.3 Grouping and Aggregation in SQL

In Section 5.2.4, we introduced the grouping-and-aggregation operator  $\gamma$  for our extended relational algebra. Recall that this operator allows us to partition

the tuples of a relation into “groups,” based on the values of tuples in one or more attributes, as discussed in Section 5.2.3. We are then able to aggregate certain other columns of the relation by applying “aggregation” operators to those columns. If there are groups, then the aggregation is done separately for each group. SQL provides all the capability of the  $\gamma$  operator through the use of aggregation operators in **SELECT** clauses and a special **GROUP BY** clause.

#### 6.4.4 Aggregation Operators

SQL uses the five aggregation operators **SUM**, **AVG**, **MIN**, **MAX**, and **COUNT** that we met in Section 5.2.2. These operators are used by applying them to a scalar-valued expression, typically a column name, in a **SELECT** clause. One exception is the expression **COUNT(\*)**, which counts all the tuples in the relation that is constructed from the **FROM** clause and **WHERE** clause of the query.

In addition, we have the option of eliminating duplicates from the column before applying the aggregation operator by using the keyword **DISTINCT**. That is, an expression such as **COUNT(DISTINCT x)** counts the number of distinct values in column  $x$ . We could use any of the other operators in place of **COUNT** here, but expressions such as **SUM(DISTINCT x)** rarely make sense, since it asks us to sum the different values in column  $x$ .

**Example 6.29:** The following query finds the average net worth of all movie executives:

```
SELECT AVG(netWorth)
  FROM MovieExec;
```

Note that there is no **WHERE** clause at all, so the keyword **WHERE** is properly omitted. This query examines the **netWorth** column of the relation

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

sums the values found there, one value for each tuple (even if the tuple is a duplicate of some other tuple), and divides the sum by the number of tuples. If there are no duplicate tuples, then this query gives the average net worth as we expect. If there were duplicate tuples, then a movie executive whose tuple appeared  $n$  times would have his or her net worth counted  $n$  times in the average. □

**Example 6.30:** The following query:

```
SELECT COUNT(*)
  FROM StarsIn;
```

counts the number of tuples in the **StarsIn** relation. The similar query:

```
SELECT COUNT(starName)
  FROM StarsIn;
```

counts the number of values in the `starName` column of the relation. Since duplicate values are not eliminated when we project onto the `starName` column in SQL, this count should be the same as the count produced by the query with `COUNT(*)`.

If we want to be certain that we do not count duplicate values more than once, we can use the keyword `DISTINCT` before the aggregated attribute, as:

```
SELECT COUNT(DISTINCT starName)
  FROM StarsIn;
```

Now, each star is counted once, no matter in how many movies they appeared.

□

#### 6.4.5 Grouping

To group tuples, we use a `GROUP BY` clause, following the `WHERE` clause. The keywords `GROUP BY` are followed by a list of *grouping* attributes. In the simplest situation, there is only one relation reference in the `FROM` clause, and this relation has its tuples grouped according to their values in the grouping attributes. Whatever aggregation operators are used in the `SELECT` clause are applied only within groups.

**Example 6.31:** The problem of finding, from the relation

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

the sum of the lengths of all movies for each studio is expressed by

```
SELECT studioName, SUM(length)
  FROM Movies
 GROUP BY studioName;
```

We may imagine that the tuples of relation `Movies` are reorganized and grouped so that all the tuples for Disney studios are together, all those for MGM are together, and so on, as was suggested in Fig. 5.4. The sums of the length components of all the tuples in each group are calculated, and for each group, the studio name is printed along with that sum. □

Observe in Example 6.31 how the `SELECT` clause has two kinds of terms. These are the only terms that may appear when there is an aggregation in the `SELECT` clause.

1. Aggregations, where an aggregate operator is applied to an attribute or expression involving attributes. As mentioned, these terms are evaluated on a per-group basis.

2. Attributes, such as `studioName` in this example, that appear in the `GROUP BY` clause. In a `SELECT` clause that has aggregations, only those attributes that are mentioned in the `GROUP BY` clause may appear unaggregated in the `SELECT` clause.

While queries involving `GROUP BY` generally have both grouping attributes and aggregations in the `SELECT` clause, it is technically not necessary to have both. For example, we could write

```
SELECT studioName
  FROM Movies
 GROUP BY studioName;
```

This query would group the tuples of `Movies` according to their studio name and then print the studio name for each group, no matter how many tuples there are with a given studio name. Thus, the above query has the same effect as

```
SELECT DISTINCT studioName
  FROM Movies;
```

It is also possible to use a `GROUP BY` clause in a query about several relations. Such a query is interpreted by the following sequence of steps:

1. Evaluate the relation  $R$  expressed by the `FROM` and `WHERE` clauses. That is, relation  $R$  is the Cartesian product of the relations mentioned in the `FROM` clause, to which the selection of the `WHERE` clause is applied.
2. Group the tuples of  $R$  according to the attributes in the `GROUP BY` clause.
3. Produce as a result the attributes and aggregations of the `SELECT` clause, as if the query were about a stored relation  $R$ .

**Example 6.32:** Suppose we wish to print a table listing each producer's total length of film produced. We need to get information from the two relations

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

so we begin by taking their theta-join, equating the certificate numbers from the two relations. That step gives us a relation in which each `MovieExec` tuple is paired with the `Movies` tuples for all the movies of that producer. Note that an executive who is not a producer will not be paired with any movies, and therefore will not appear in the relation. Now, we can group the selected tuples of this relation according to the name of the producer. Finally, we sum the lengths of the movies in each group. The query is shown in Fig. 6.13. □

```

SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name;

```

Figure 6.13: Computing the length of movies for each producer

### 6.4.6 Grouping, Aggregation, and Nulls

When tuples have nulls, there are a few rules we must remember:

- The value **NULL** is ignored in any aggregation. It does not contribute to a sum, average, or count of an attribute, nor can it be the minimum or maximum in its column. For example, **COUNT(\*)** is always a count of the number of tuples in a relation, but **COUNT(A)** is the number of tuples with non-**NULL** values for attribute *A*.
- On the other hand, **NULL** is treated as an ordinary value when forming groups. That is, we can have a group in which one or more of the grouping attributes are assigned the value **NULL**.
- When we perform any aggregation except count over an empty bag of values, the result is **NULL**. The count of an empty bag is 0.

**Example 6.33:** Suppose we have a relation  $R(A, B)$  with one tuple, both of whose components are **NULL**:

<i>A</i>	<i>B</i>
NULL	NULL

Then the result of:

```

SELECT A, COUNT(B)
FROM R
GROUP BY A;

```

is the one tuple  $(\text{NULL}, 0)$ . The reason is that when we group by *A*, we find only a group for value **NULL**. This group has one tuple, and its *B*-value is **NULL**. We thus count the bag of values  $\{\text{NULL}\}$ . Since the count of a bag of values does not count the **NULL**'s, this count is 0.

On the other hand, the result of:

```

SELECT A, SUM(B)
FROM R
GROUP BY A;

```

## Order of Clauses in SQL Queries

We have now met all six clauses that can appear in a SQL “select-from-where” query: **SELECT**, **FROM**, **WHERE**, **GROUP BY**, **HAVING**, and **ORDER BY**. Only the **SELECT** and **FROM** clauses are required. Whichever additional clauses appear must be in the order listed above.

is the one tuple  $(\text{NULL}, \text{NULL})$ . The reason is as follows. The group for value  $\text{NULL}$  has one tuple, the only tuple in  $R$ . However, when we try to sum the  $B$ -values for this group, we only find  $\text{NULL}$ , and  $\text{NULL}$  does not contribute to a sum. Thus, we are summing an empty bag of values, and this sum is defined to be  $\text{NULL}$ .  $\square$

### 6.4.7 HAVING Clauses

Suppose that we did not wish to include all of the producers in our table of Example 6.32. We could restrict the tuples prior to grouping in a way that would make undesired groups empty. For instance, if we only wanted the total length of movies for producers with a net worth of more than \$10,000,000, we could change the third line of Fig. 6.13 to

```
WHERE producerC# = cert# AND networth > 10000000
```

However, sometimes we want to choose our groups based on some aggregate property of the group itself. Then we follow the **GROUP BY** clause with a **HAVING** clause. The latter clause consists of the keyword **HAVING** followed by a condition about the group.

**Example 6.34:** Suppose we want to print the total film length for only those producers who made at least one film prior to 1930. We may append to Fig. 6.13 the clause

```
HAVING MIN(year) < 1930
```

The resulting query, shown in Fig. 6.14, would remove from the grouped relation all those groups in which every tuple had a **year** component 1930 or higher.  
 $\square$

There are several rules we must remember about **HAVING** clauses:

- An aggregation in a **HAVING** clause applies only to the tuples of the group being tested.
- Any attribute of relations in the **FROM** clause may be aggregated in the **HAVING** clause, but only those attributes that are in the **GROUP BY** list may appear unaggregated in the **HAVING** clause (the same rule as for the **SELECT** clause).

```

SELECT name, SUM(length)
FROM MovieExec, Movies
WHERE producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;

```

Figure 6.14: Computing the total length of film for early producers

### 6.4.8 Exercises for Section 6.4

**Exercise 6.4.1:** Write each of the queries in Exercise 2.4.1 in SQL, making sure that duplicates are eliminated.

**Exercise 6.4.2:** Write each of the queries in Exercise 2.4.3 in SQL, making sure that duplicates are eliminated.

! **Exercise 6.4.3:** For each of your answers to Exercise 6.3.1, determine whether or not the result of your query can have duplicates. If so, rewrite the query to eliminate duplicates. If not, write a query without subqueries that has the same, duplicate-free answer.

! **Exercise 6.4.4:** Repeat Exercise 6.4.3 for your answers to Exercise 6.3.2.

! **Exercise 6.4.5:** In Example 6.27, we mentioned that different versions of the query “find the producers of Harrison Ford’s movies” can have different answers as bags, even though they yield the same set of answers. Consider the version of the query in Example 6.22, where we used a subquery in the FROM clause. Does this version produce duplicates, and if so, why?

**Exercise 6.4.6:** Write the following queries, based on the database schema

```

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

```

of Exercise 2.4.1, and evaluate your queries using the data of that exercise.

- a) Find the average speed of PC’s.
- b) Find the average speed of laptops costing over \$1000.
- c) Find the average price of PC’s made by manufacturer “A.”
- ! d) Find the average price of PC’s and laptops made by manufacturer “D.”
- e) Find, for each different speed, the average price of a PC.

- ! f) Find for each manufacturer, the average screen size of its laptops.
- ! g) Find the manufacturers that make at least three different models of PC.
- ! h) Find for each manufacturer who sells PC's the maximum price of a PC.
- ! i) Find, for each speed of PC above 2.0, the average price.
- !! j) Find the average hard disk size of a PC for all those manufacturers that make printers.

**Exercise 6.4.7:** Write the following queries, based on the database schema

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

of Exercise 2.4.3, and evaluate your queries using the data of that exercise.

- a) Find the number of battleship classes.
- b) Find the average number of guns of battleship classes.
- ! c) Find the average number of guns of battleships. Note the difference between (b) and (c); do we weight a class by the number of ships of that class or not?
- ! d) Find for each class the year in which the first ship of that class was launched.
- ! e) Find for each class the number of ships of that class sunk in battle.
- !! f) Find for each class with at least three ships the number of ships of that class sunk in battle.
- !! g) The weight (in pounds) of the shell fired from a naval gun is approximately one half the cube of the bore (in inches). Find the average weight of the shell for each country's ships.

**Exercise 6.4.8:** In Example 5.10 we gave an example of the query: “find, for each star who has appeared in at least three movies, the earliest year in which they appeared.” We wrote this query as a  $\gamma$  operation. Write it in SQL.

- ! **Exercise 6.4.9:** The  $\gamma$  operator of extended relational algebra does not have a feature that corresponds to the HAVING clause of SQL. Is it possible to mimic a SQL query with a HAVING clause in relational algebra? If so, how would we do it in general?

## 6.5 Database Modifications

To this point, we have focused on the normal SQL query form: the select-from-where statement. There are a number of other statement forms that do not return a result, but rather change the state of the database. In this section, we shall focus on three types of statements that allow us to

1. Insert tuples into a relation.
2. Delete certain tuples from a relation.
3. Update values of certain components of certain existing tuples.

We refer to these three types of operations collectively as *modifications*.

### 6.5.1 Insertion

The basic form of insertion statement is:

```
INSERT INTO R(A1, ..., An) VALUES (v1, ..., vn);
```

A tuple is created using the value  $v_i$  for attribute  $A_i$ , for  $i = 1, 2, \dots, n$ . If the list of attributes does not include all attributes of the relation  $R$ , then the tuple created has default values for all missing attributes.

**Example 6.35:** Suppose we wish to add Sydney Greenstreet to the list of stars of *The Maltese Falcon*. We say:

- 1) INSERT INTO StarsIn(movieTitle, movieYear, starName)
- 2) VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');

The effect of executing this statement is that a tuple with the three components on line (2) is inserted into the relation **StarsIn**. Since all attributes of **StarsIn** are mentioned on line (1), there is no need to add default components. The values on line (2) are matched with the attributes on line (1) in the order given, so 'The Maltese Falcon' becomes the value of the component for attribute **movieTitle**, and so on.  $\square$

If, as in Example 6.35, we provide values for all attributes of the relation, then we may omit the list of attributes that follows the relation name. That is, we could just say:

```
INSERT INTO StarsIn
VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');
```

However, if we take this option, we must be sure that the order of the values is the same as the standard order of attributes for the relation.

- If you are not sure of the declared order for the attributes, it is best to list them in the `INSERT` clause in the order you choose for their values in the `VALUES` clause.

The simple `INSERT` described above only puts one tuple into a relation. Instead of using explicit values for one tuple, we can compute a set of tuples to be inserted, using a subquery. This subquery replaces the keyword `VALUES` and the tuple expression in the `INSERT` statement form described above.

**Example 6.36:** Suppose we want to add to the relation

```
Studio(name, address, presC#)
```

all movie studios that are mentioned in the relation

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

but do not appear in `Studio`. Since there is no way to determine an address or a president for such a studio, we shall have to be content with value `NULL` for attributes `address` and `presC#` in the inserted `Studio` tuples. A way to make this insertion is shown in Fig. 6.15.

```

1)  INSERT INTO Studio(name)
2)    SELECT DISTINCT studioName
3)      FROM Movies
4)     WHERE studioName NOT IN
5)          (SELECT name
6)            FROM Studio);
```

Figure 6.15: Adding new studios

Like most SQL statements with nesting, Fig. 6.15 is easiest to examine from the inside out. Lines (5) and (6) generate all the studio names in the relation `Studio`. Thus, line (4) tests that a studio name from the `Movies` relation is none of these studios.

Now, we see that lines (2) through (6) produce the set of studio names found in `Movies` but not in `Studio`. The use of `DISTINCT` on line (2) assures that each studio will appear only once in this set, no matter how many movies it owns. Finally, line (1) inserts each of these studios, with `NULL` for the attributes `address` and `presC#`, into relation `Studio`. □

## 6.5.2 Deletion

The form of a deletion is

```
DELETE FROM R WHERE <condition>;
```

## The Timing of Insertions

The SQL standard requires that the query be evaluated completely before any tuples are inserted. For example, in Fig. 6.15, the query of lines (2) through (6) must be evaluated prior to executing the insertion of line (1). Thus, there is no possibility that new tuples added to `Studio` at line (1) will affect the condition on line (4).

In this particular example, it does not matter whether or not insertions are delayed until the query is completely evaluated. However, suppose `DISTINCT` were removed from line (2) of Fig. 6.15. If we evaluate the query of lines (2) through (6) before doing any insertion, then a new studio name appearing in several `Movies` tuples would appear several times in the result of this query and therefore would be inserted several times into relation `Studio`. However, if the DBMS inserted new studios into `Studio` as soon as we found them during the evaluation of the query of lines (2) through (6), something that would be incorrect according to the standard, then the same new studio would not be inserted twice. Rather, as soon as the new studio was inserted once, its name would no longer satisfy the condition of lines (4) through (6), and it would not appear a second time in the result of the query of lines (2) through (6).

The effect of executing this statement is that every tuple satisfying the condition will be deleted from relation  $R$ .

**Example 6.37:** We can delete from relation

```
StarsIn(movieTitle, movieYear, starName)
```

the fact that Sydney Greenstreet was a star in *The Maltese Falcon* by the SQL statement:

```
DELETE FROM StarsIn
WHERE movieTitle = 'The Maltese Falcon' AND
      movieYear = 1942 AND
      starName = 'Sydney Greenstreet';
```

Notice that unlike the insertion statement of Example 6.35, we cannot simply specify a tuple to be deleted. Rather, we must describe the tuple exactly by a `WHERE` clause.  $\square$

**Example 6.38:** Here is another example of a deletion. This time, we delete from relation

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

several tuples at once by using a condition that can be satisfied by more than one tuple. The statement

```
DELETE FROM MovieExec
WHERE netWorth < 10000000;
```

deletes all movie executives whose net worth is low — less than ten million dollars. □

### 6.5.3 Updates

While we might think of both insertions and deletions of tuples as “updates” to the database, an *update* in SQL is a very specific kind of change to the database: one or more tuples that already exist in the database have some of their components changed. The general form of an update statement is:

```
UPDATE R SET <new-value assignments> WHERE <condition>;
```

Each new-value assignment is an attribute, an equal sign, and an expression. If there is more than one assignment, they are separated by commas. The effect of this statement is to find all the tuples in *R* that satisfy the condition. Each of these tuples is then changed by having the expressions in the assignments evaluated and assigned to the components of the tuple for the corresponding attributes of *R*.

**Example 6.39:** Let us modify the relation

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

by attaching the title **Pres.** in front of the name of every movie executive who is the president of a studio. The condition the desired tuples satisfy is that their certificate numbers appear in the **presC#** component of some tuple in the **Studio** relation. We express this update as:

- 1) UPDATE MovieExec
- 2) SET name = 'Pres. ' || name
- 3) WHERE cert# IN (SELECT presC# FROM Studio);

Line (3) tests whether the certificate number from the **MovieExec** tuple is one of those that appear as a president’s certificate number in **Studio**.

Line (2) performs the update on the selected tuples. Recall that the operator **||** denotes concatenation of strings, so the expression following the **=** sign in line (2) places the characters **Pres.** and a blank in front of the old value of the **name** component of this tuple. The new string becomes the value of the **name** component of this tuple; the effect is that '**Pres.**' has been prepended to the old value of **name**. □

### 6.5.4 Exercises for Section 6.5

**Exercise 6.5.1:** Write the following database modifications, based on the database schema

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

of Exercise 2.4.1. Describe the effect of the modifications on the data of that exercise.

- a) Using two **INSERT** statements, store in the database the fact that PC model 1100 is made by manufacturer C, has speed 3.2, RAM 1024, hard disk 180, and sells for \$2499.
- b) Insert the facts that for every PC there is a laptop with the same manufacturer, speed, RAM, and hard disk, a 17-inch screen, a model number 1100 greater, and a price \$500 more.
- c) Delete all PC's with less than 100 gigabytes of hard disk.
- d) Delete all laptops made by a manufacturer that doesn't make printers.
- e) Manufacturer A buys manufacturer B. Change all products made by B so they are now made by A.
- f) For each PC, double the amount of RAM and add 60 gigabytes to the amount of hard disk. (Remember that several attributes can be changed by one **UPDATE** statement.)
- g) For each laptop made by manufacturer B, add one inch to the screen size and subtract \$100 from the price.

**Exercise 6.5.2:** Write the following database modifications, based on the database schema

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

of Exercise 2.4.3. Describe the effect of the modifications on the data of that exercise.

- a) The two British battleships of the Nelson class — Nelson and Rodney — were both launched in 1927, had nine 16-inch guns, and a displacement of 34,000 tons. Insert these facts into the database.

- b) Two of the three battleships of the Italian Vittorio Veneto class — Vittorio Veneto and Italia — were launched in 1940; the third ship of that class, Roma, was launched in 1942. Each had nine 15-inch guns and a displacement of 41,000 tons. Insert these facts into the database.
- c) Delete from Ships all ships sunk in battle.
- d) Modify the **Classes** relation so that gun bores are measured in centimeters (one inch = 2.5 centimeters) and displacements are measured in metric tons (one metric ton = 1.1 tons).
- e) Delete all classes with fewer than three ships.

## 6.6 Transactions in SQL

To this point, our model of operations on the database has been that of one user querying or modifying the database. Thus, operations on the database are executed one at a time, and the database state left by one operation is the state upon which the next operation acts. Moreover, we imagine that operations are carried out in their entirety (“atomically”). That is, we assumed it is impossible for the hardware or software to fail in the middle of a modification, leaving the database in a state that cannot be explained as the result of the operations performed on it.

Real life is often considerably more complicated. We shall first consider what can happen to leave the database in a state that doesn’t reflect the operations performed on it, and then we shall consider the tools SQL gives the user to assure that these problems do not occur.

### 6.6.1 Serializability

In applications like Web services, banking, or airline reservations, hundreds of operations per second may be performed on the database. The operations initiate at any of thousands or millions of sites, such as desktop computers or automatic teller machines. It is entirely possible that we could have two operations affecting the same bank account or flight, and for those operations to overlap in time. If so, they might interact in strange ways.

Here is an example of what could go wrong if the DBMS were completely unconstrained as to the order in which it operated upon the database. This example involves a database interacting with people, and it is intended to illustrate why it is important to control the sequences in which interacting events can occur. However, a DBMS would not control events that were so “large” that they involved waiting for a user to make a choice. The event sequences controlled by the DBMS involve only the execution of SQL statements.

**Example 6.40:** The typical airline gives customers a Web interface where they can choose a seat for their flight. This interface shows a map of available

seats, and the data for this map is obtained from the airline's database. There might be a relation such as:

```
Flights(fltNo, fltDate, seatNo, seatStatus)
```

upon which we can issue the query:

```
SELECT seatNo  
FROM Flights  
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'  
      AND seatStatus = 'available';
```

The flight number and date are example data, which would in fact be obtained from previous interactions with the customer.

When the customer clicks on an empty seat, say 22A, that seat is reserved for them. The database is modified by an update-statement, such as:

```
UPDATE Flights  
SET seatStatus = 'occupied'  
WHERE fltNo = 123 AND fltDate = DATE '2008-12-25'  
      AND seatNo = '22A';
```

However, this customer may not be the only one reserving a seat on flight 123 on Dec. 25, 2008 and this exact moment. Another customer may have asked for the seat map at the same time, in which case they also see seat 22A empty. Should they also choose seat 22A, they too believe they have reserved 22A. The timing of these events is as suggested by Fig. 6.16. □

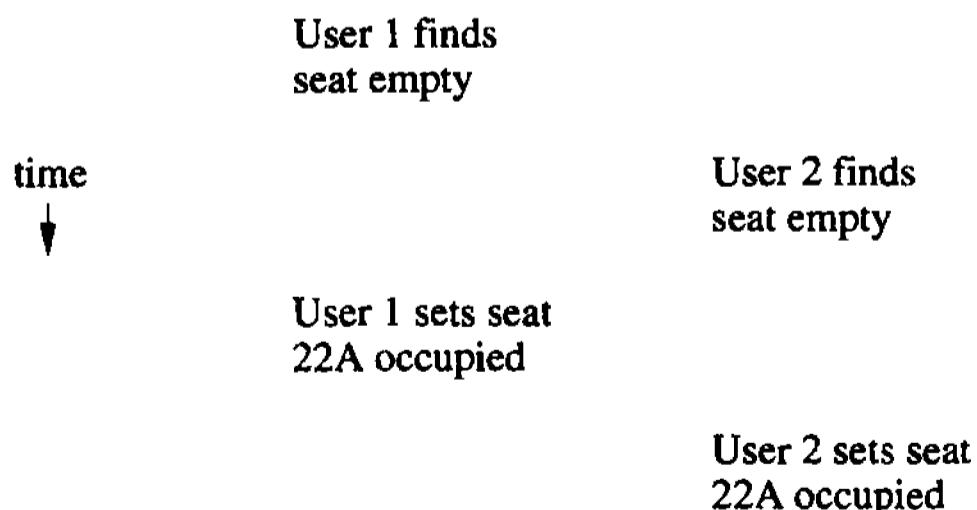


Figure 6.16: Two customers trying to book the same seat simultaneously

As we see from Example 6.40, it is conceivable that two operations could each be performed correctly, and yet the global result not be correct: both customers believe they have been granted seat 22A. The problem is solved in SQL by the notion of a “transaction,” which is informally a group of operations that need to be performed together. Suppose that in Example 6.40, the query

## Assuring Serializable Behavior

In practice it is often impossible to require that operations run serially; there are just too many of them, and some parallelism is required. Thus, DBMS's adopt a mechanism for assuring serializable behavior; even if the execution is not serial, the result looks to users as if operations were executed serially.

One common approach is for the DBMS to *lock* elements of the database so that two functions cannot access them at the same time. We mentioned locking in Section 1.2.4, and there is an extensive technology of how to implement locks in a DBMS. For example, if the transaction of Example 6.40 were written to lock other transactions out of the `Flights` relation, then transactions that did not access `Flights` could run in parallel with the seat-selection transaction, but no other invocation of the seat-selection operation could run in parallel.

and update shown would be grouped into one transaction.<sup>6</sup> SQL then allows the programmer to state that a certain transaction must be *serializable* with respect to other transactions. That is, these transactions must behave as if they were run *serially* — one at a time, with no overlap.

Clearly, if the two invocations of the seat-selection operation are run serially (or serializably), then the error we saw cannot occur. One customer's invocation occurs first. This customer sees seat 22A is empty, and books it. The other customer's invocation then begins and is not given 22A as a choice, because it is already occupied. It may matter to the customers who gets the seat, but to the database all that is important is that a seat is assigned only once.

### 6.6.2 Atomicity

In addition to nonserialized behavior that can occur if two or more database operations are performed about the same time, it is possible for a single operation to put the database in an unacceptable state if there is a hardware or software “crash” while the operation is executing. Here is another example suggesting what might occur. As in Example 6.40, we should remember that real database systems do not allow this sort of error to occur in properly designed application programs.

**Example 6.41:** Let us picture another common sort of database: a bank's account records. We can represent the situation by a relation

<sup>6</sup>However, it would be extremely unwise to group into a single transaction operations that involved a user, or even a computer that was not owned by the airline, such as a travel agent's computer. Another mechanism must be used to deal with event sequences that include operations outside the database.

**Accounts(acctNo, balance)**

Consider the operation of transferring \$100 from the account numbered 123 to the account 456. We might first check whether there is at least \$100 in account 123, and if so, we execute the following two steps:

1. Add \$100 to account 456 by the SQL update statement:

```
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

2. Subtract \$100 from account 123 by the SQL update statement:

```
UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

Now, consider what happens if there is a failure after Step (1) but before Step (2). Perhaps the computer fails, or the network connecting the database to the processor that is actually performing the transfer fails. Then the database is left in a state where money has been transferred into the second account, but the money has not been taken out of the first account. The bank has in effect given away the amount of money that was to be transferred. □

The problem illustrated by Example 6.41 is that certain combinations of database operations, like the two updates of that example, need to be done *atomically*; that is, either they are both done or neither is done. For example, a simple solution is to have all changes to the database done in a local workspace, and only after all work is done do we *commit* the changes to the database, whereupon all changes become part of the database and visible to other operations.

### 6.6.3 Transactions

The solution to the problems of serialization and atomicity posed in Sections 6.6.1 and 6.6.2 is to group database operations into *transactions*. A transaction is a collection of one or more operations on the database that must be executed atomically; that is, either all operations are performed or none are. In addition, SQL requires that, as a default, transactions are executed in a serializable manner. A DBMS may allow the user to specify a less stringent constraint on the interleaving of operations from two or more transactions. We shall discuss these modifications to the serializability condition in later sections.

When using the *generic SQL interface* (the facility wherein one types queries and other SQL statements), each statement is a transaction by itself. However,

## How the Database Changes During Transactions

Different systems may do different things to implement transactions. It is possible that as a transaction executes, it makes changes to the database. If the transaction aborts, then (unless the programmer took precautions) it is possible that these changes were seen by some other transaction. The most common solution is for the database system to lock the changed items until `COMMIT` or `ROLLBACK` is chosen, thus preventing other transactions from seeing the tentative change. Locks or an equivalent would surely be used if the user wants the transactions to run in a serializable fashion.

However, as we shall see starting in Section 6.6.4, SQL offers us several options regarding the treatment of tentative database changes. It is possible that the changed data is not locked and becomes visible even though a subsequent rollback makes the change disappear. It is up to the author of a transaction to decide whether it is safe for that transaction to see tentative changes of other transactions.

SQL allows the programmer to group several statements into a single transaction. The SQL command `START TRANSACTION` is used to mark the beginning of a transaction. There are two ways to end a transaction:

1. The SQL statement `COMMIT` causes the transaction to end successfully. Whatever changes to the database were caused by the SQL statement or statements since the current transaction began are installed permanently in the database (i.e., they are *committed*). Before the `COMMIT` statement is executed, changes are tentative and may or may not be visible to other transactions.
2. The SQL statement `ROLLBACK` causes the transaction to *abort*, or terminate unsuccessfully. Any changes made in response to the SQL statements of the transaction are undone (i.e., they are *rolled back*), so they never permanently appear in the database.

**Example 6.42:** Suppose we want the transfer operation of Example 6.41 to be a single transaction. We execute `BEGIN TRANSACTION` before accessing the database. If we find that there are insufficient funds to make the transfer, then we would execute the `ROLLBACK` command. However, if there are sufficient funds, then we execute the two update statements and then execute `COMMIT`. □

### 6.6.4 Read-Only Transactions

Examples 6.40 and 6.41 each involved a transaction that read and then (possibly) wrote some data into the database. This sort of transaction is prone to

## Application- Versus System-Generated Rollbacks

In our discussion of transactions, we have presumed that the decision whether a transaction is committed or rolled back is made as part of the application issuing the transaction. That is, as in Examples 6.44 and 6.42, a transaction may perform a number of database operations, then decide whether to make any changes permanent by issuing COMMIT, or to return to the original state by issuing ROLLBACK. However, the system may also perform transaction rollbacks, to ensure that transactions are executed atomically and conform to their specified isolation level in the presence of other concurrent transactions or system crashes. Typically, if the system aborts a transaction then a special error code or exception is generated. If an application wishes to guarantee that its transactions are executed successfully, it must catch such conditions and reissue the transaction in question.

serialization problems. Thus we saw in Example 6.40 what could happen if two executions of the function tried to book the same seat at the same time, and we saw in Example 6.41 what could happen if there was a crash in the middle of a funds transfer. However, when a transaction only reads data and does not write data, we have more freedom to let the transaction execute in parallel with other transactions.

**Example 6.43:** Suppose we wrote a program that read data from the Flights relation of Example 6.40 to determine whether a certain seat was available. We could execute many invocations of this program at once, without risk of permanent harm to the database. The worst that could happen is that while we were reading the availability of a certain seat, that seat was being booked or was being released by the execution of some other program. Thus, we might get the answer “available” or “occupied,” depending on microscopic differences in the time at which we executed the query, but the answer would make sense at some time. □

If we tell the SQL execution system that our current transaction is *read-only*, that is, it will never change the database, then it is quite possible that the SQL system will be able to take advantage of that knowledge. Generally it will be possible for many read-only transactions accessing the same data to run in parallel, while they would not be allowed to run in parallel with a transaction that wrote the same data.

We tell the SQL system that the next transaction is read-only by:

```
SET TRANSACTION READ ONLY;
```

This statement must be executed before the transaction begins. We can also inform SQL that the coming transaction may write data by the statement

```
SET TRANSACTION READ WRITE;
```

However, this option is the default.

### 6.6.5 Dirty Reads

*Dirty data* is a common term for data written by a transaction that has not yet committed. A *dirty read* is a read of dirty data written by another transaction. The risk in reading dirty data is that the transaction that wrote it may eventually abort. If so, then the dirty data will be removed from the database, and the world is supposed to behave as if that data never existed. If some other transaction has read the dirty data, then that transaction might commit or take some other action that reflects its knowledge of the dirty data.

Sometimes the dirty read matters, and sometimes it doesn't. Other times it matters little enough that it makes sense to risk an occasional dirty read and thus avoid:

1. The time-consuming work by the DBMS that is needed to prevent dirty reads, and
2. The loss of parallelism that results from waiting until there is no possibility of a dirty read.

Here are some examples of what might happen when dirty reads are allowed.

**Example 6.44:** Let us reconsider the account transfer of Example 6.41. However, suppose that transfers are implemented by a program  $P$  that executes the following sequence of steps:

1. Add money to account 2.
2. Test if account 1 has enough money.
  - (a) If there is not enough money, remove the money from account 2 and end.<sup>7</sup>
  - (b) If there is enough money, subtract the money from account 1 and end.

If program  $P$  is executed serializably, then it doesn't matter that we have put money temporarily into account 2. No one will see that money, and it gets removed if the transfer can't be made.

However, suppose dirty reads are possible. Imagine there are three accounts:  $A_1$ ,  $A_2$ , and  $A_3$ , with \$100, \$200, and \$300, respectively. Suppose transaction

---

<sup>7</sup>You should be aware that the program  $P$  is trying to perform functions that would more typically be done by the DBMS. In particular, when  $P$  decides, as it has done at this step, that it must not complete the transaction, it would issue a rollback (abort) command to the DBMS and have the DBMS reverse the effects of this execution of  $P$ .

$T_1$  executes program  $P$  to transfer \$150 from  $A_1$  to  $A_2$ . At roughly the same time, transaction  $T_2$  runs program  $P$  to transfer \$250 from  $A_2$  to  $A_3$ . Here is a possible sequence of events:

1.  $T_2$  executes Step (1) and adds \$250 to  $A_3$ , which now has \$550.
2.  $T_1$  executes Step (1) and adds \$150 to  $A_2$ , which now has \$350.
3.  $T_2$  executes the test of Step (2) and finds that  $A_2$  has enough funds (\$350) to allow the transfer of \$250 from  $A_2$  to  $A_3$ .
4.  $T_1$  executes the test of Step (2) and finds that  $A_1$  does not have enough funds (\$100) to allow the transfer of \$150 from  $A_1$  to  $A_2$ .
5.  $T_2$  executes Step (2b). It subtracts \$250 from  $A_2$ , which now has \$100, and ends.
6.  $T_1$  executes Step (2a). It subtracts \$150 from  $A_2$ , which now has -\$50, and ends.

The total amount of money has not changed; there is still \$600 among the three accounts. But because  $T_2$  read dirty data at the third of the six steps above, we have not protected against an account going negative, which supposedly was the purpose of testing the first account to see if it had adequate funds.  $\square$

**Example 6.45:** Let us imagine a variation on the seat-choosing function of Example 6.40. In the new approach:

1. We find an available seat and reserve it by setting `seatStatus` to 'occupied' for that seat. If there is none, end.
2. We ask the customer for approval of the seat. If so, we commit. If not, we release the seat by setting `seatStatus` to 'available' and repeat Step (1) to get another seat.

If two transactions are executing this algorithm at about the same time, one might reserve a seat  $S$ , which later is rejected by the customer. If the second transaction executes Step (1) at a time when seat  $S$  is marked occupied, the customer for that transaction is not given the option to take seat  $S$ .

As in Example 6.44, the problem is that a dirty read has occurred. The second transaction saw a tuple (with  $S$  marked occupied) that was written by the first transaction and later modified by the first transaction.  $\square$

How important is the fact that a read was dirty? In Example 6.44 it was very important; it caused an account to go negative despite apparent safeguards against that happening. In Example 6.45, the problem does not look too serious. Indeed, the second traveler might not get their favorite seat, or might even be told that no seats existed. However, in the latter case, running the transaction

again will almost certainly reveal the availability of seat  $S$ . It might well make sense to implement this seat-choosing function in a way that allowed dirty reads, in order to speed up the average processing time for booking requests.

SQL allows us to specify that dirty reads are acceptable for a given transaction. We use the **SET TRANSACTION** statement that we discussed in Section 6.6.4. The appropriate form for a transaction like that described in Example 6.45 is:

```
1) SET TRANSACTION READ WRITE
2)      ISOLATION LEVEL READ UNCOMMITTED;
```

The statement above does two things:

1. Line (1) declares that the transaction may write data.
2. Line (2) declares that the transaction may run with the “isolation level” *read-uncommitted*. That is, the transaction is allowed to read dirty data. We shall discuss the four isolation levels in Section 6.6.6. So far, we have seen two of them: *serializable* and *read-uncommitted*.

Note that if the transaction is not read-only (i.e., it may modify the database), and we specify isolation level **READ UNCOMMITTED**, then we must also specify **READ WRITE**. Recall from Section 6.6.4 that the default assumption is that transactions are read-write. However, SQL makes an exception for the case where dirty reads are allowed. Then, the default assumption is that the transaction is read-only, because read-write transactions with dirty reads entail significant risks, as we saw. If we want a read-write transaction to run with *read-uncommitted* as the isolation level, then we need to specify **READ WRITE** explicitly, as above.

### 6.6.6 Other Isolation Levels

SQL provides a total of four *isolation levels*. Two of them we have already seen: *serializable* and *read-uncommitted* (dirty reads allowed). The other two are *read-committed* and *repeatable-read*. They can be specified for a given transaction by

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

or

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

respectively. For each, the default is that transactions are read-write, so we can add **READ ONLY** to either statement, if appropriate. Incidentally, we also have the option of specifying

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

## Interactions Among Transactions Running at Different Isolation Levels

A subtle point is that the isolation level of a transaction affects only what data *that* transaction may see; it does not affect what any other transaction sees. As a case in point, if a transaction  $T$  is running at level serializable, then the execution of  $T$  must appear as if all other transactions run either entirely before or entirely after  $T$ . However, if some of those transactions are running at another isolation level, then *they* may see the data written by  $T$  as  $T$  writes it. They may even see dirty data from  $T$  if they are running at isolation level read-uncommitted, and  $T$  aborts.

However, that is the SQL default and need not be stated explicitly.

The read-committed isolation level, as its name implies, forbids the reading of dirty (uncommitted) data. However, it does allow a transaction running at this isolation level to issue the same query several times and get different answers, as long as the answers reflect data that has been written by transactions that already committed.

**Example 6.46:** Let us reconsider the seat-choosing program of Example 6.45, but suppose we declare it to run with isolation level read-committed. Then when it searches for a seat at Step (1), it will not see seats as booked if some other transaction is reserving them but not committed.<sup>8</sup> However, if the traveler rejects seats, and one execution of the function queries for available seats many times, it may see a different set of available seats each time it queries, as other transactions successfully book seats or cancel seats in parallel with our transaction. □

Now, let us consider isolation level repeatable-read. The term is something of a misnomer, since the same query issued more than once is not quite guaranteed to get the same answer. Under repeatable-read isolation, if a tuple is retrieved the first time, then we can be sure that the identical tuple will be retrieved again if the query is repeated. However, it is also possible that a second or subsequent execution of the same query will retrieve *phantom* tuples. The latter are tuples that result from insertions into the database while our transaction is executing.

**Example 6.47:** Let us continue with the seat-choosing problem of Examples 6.45 and 6.46. If we execute this function under isolation level repeatable-read,

<sup>8</sup>What actually happens may seem mysterious, since we have not addressed the algorithms for enforcing the various isolation levels. Possibly, should two transactions both see a seat as available and try to book it, one will be forced by the system to roll back in order to break the deadlock (see the box on “Application- Versus System-Generated Rollbacks” in Section 6.6.3).

then a seat that is available on the first query at Step (1) will remain available at subsequent queries.

However, suppose some new tuples enter the relation **Flights**. For example, the airline may have switched the flight to a larger plane, creating some new tuples that weren't there before. Then under repeatable-read isolation, a subsequent query for available seats may also retrieve the new seats. □

Figure 6.17 summarizes the differences between the four SQL isolation levels.

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed

Figure 6.17: Properties of SQL isolation levels

### 6.6.7 Exercises for Section 6.6

**Exercise 6.6.1:** This and the next exercises involve certain programs that operate on the two relations

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
```

from our running PC exercise. Sketch the following programs, including SQL statements and work done in a conventional language. Do not forget to issue **BEGIN TRANSACTION**, **COMMIT**, and **ROLLBACK** statements at the proper times and to tell the system your transactions are read-only if they are.

- a) Given a speed and amount of RAM (as arguments of the function), look up the PC's with that speed and RAM, printing the model number and price of each.
- b) Given a model number, delete the tuple for that model from both PC and Product.
- c) Given a model number, decrease the price of that model PC by \$100.
- d) Given a maker, model number, processor speed, RAM size, hard-disk size, and price, check that there is no product with that model. If there is such a model, print an error message for the user. If no such model existed in the database, enter the information about that model into the PC and Product tables.

- ! **Exercise 6.6.2:** For each of the programs of Exercise 6.6.1, discuss the atomicity problems, if any, that could occur should the system crash in the middle of an execution of the program.
- ! **Exercise 6.6.3:** Suppose we execute as a transaction  $T$  one of the four programs of Exercise 6.6.1, while other transactions that are executions of the same or a different one of the four programs may also be executing at about the same time. What behaviors of transaction  $T$  may be observed if all the transactions run with isolation level **READ UNCOMMITTED** that would not be possible if they all ran with isolation level **SERIALIZABLE**? Consider separately the case that  $T$  is any of the programs (a) through (d) of Exercise 6.6.1.
- !! **Exercise 6.6.4:** Suppose we have a transaction  $T$  that is a function which runs “forever,” and at each hour checks whether there is a PC that has a speed of 3.5 or more and sells for under \$1000. If it finds one, it prints the information and terminates. During this time, other transactions that are executions of one of the four programs described in Exercise 6.6.1 may run. For each of the four isolation levels — serializable, repeatable read, read committed, and read uncommitted — tell what the effect on  $T$  of running at this isolation level is.

## 6.7 Summary of Chapter 6

- ◆ **SQL:** The language SQL is the principal query language for relational database systems. The most recent full standard is called SQL-99 or SQL3. Commercial systems generally vary from this standard.
- ◆ **Select-From-Where Queries:** The most common form of SQL query has the form select-from-where. It allows us to take the product of several relations (the **FROM** clause), apply a condition to the tuples of the result (the **WHERE** clause), and produce desired components (the **SELECT** clause).
- ◆ **Subqueries:** Select-from-where queries can also be used as subqueries within a **WHERE** clause or **FROM** clause of another query. The operators **EXISTS**, **IN**, **ALL**, and **ANY** may be used to express boolean-valued conditions about the relations that are the result of a subquery in a **WHERE** clause.
- ◆ **Set Operations on Relations:** We can take the union, intersection, or difference of relations by connecting the relations, or connecting queries defining the relations, with the keywords **UNION**, **INTERSECT**, and **EXCEPT**, respectively.
- ◆ **Join Expressions:** SQL has operators such as **NATURAL JOIN** that may be applied to relations, either as queries by themselves or to define relations in a **FROM** clause.

- ◆ *Null Values*: SQL provides a special value **NULL** that appears in components of tuples for which no concrete value is available. The arithmetic and logic of **NULL** is unusual. Comparison of any value to **NULL**, even another **NULL**, gives the truth value **UNKNOWN**. That truth value, in turn, behaves in boolean-valued expressions as if it were halfway between **TRUE** and **FALSE**.
- ◆ *Outerjoins*: SQL provides an **OUTER JOIN** operator that joins relations but also includes in the result dangling tuples from one or both relations; the dangling tuples are padded with **NULL**'s in the resulting relation.
- ◆ *The Bag Model of Relations*: SQL actually regards relations as bags of tuples, not sets of tuples. We can force elimination of duplicate tuples with the keyword **DISTINCT**, while keyword **ALL** allows the result to be a bag in certain circumstances where bags are not the default.
- ◆ *Aggregations*: The values appearing in one column of a relation can be summarized (aggregated) by using one of the keywords **SUM**, **AVG** (average value), **MIN**, **MAX**, or **COUNT**. Tuples can be partitioned prior to aggregation with the keywords **GROUP BY**. Certain groups can be eliminated with a clause introduced by the keyword **HAVING**.
- ◆ *Modification Statements*: SQL allows us to change the tuples in a relation. We may **INSERT** (add new tuples), **DELETE** (remove tuples), or **UPDATE** (change some of the existing tuples), by writing SQL statements using one of these three keywords.
- ◆ *Transactions*: SQL allows the programmer to group SQL statements into transactions, which may be committed or rolled back (aborted). Transactions may be rolled back by the application in order to undo changes, or by the system in order to guarantee atomicity and isolation.
- ◆ *Isolation Levels*: SQL defines four isolation levels called, from most stringent to least stringent: “*serializable*” (the transaction must appear to run either completely before or completely after each other transaction), “*repeatable-read*” (every tuple read in response to a query will reappear if the query is repeated), “*read-committed*” (only tuples written by transactions that have already committed may be seen by this transaction), and “*read-uncommitted*” (no constraint on what the transaction may see).

## 6.8 References for Chapter 6

Many books on SQL programming are available. Some popular ones are [3], [5], and [7]. [6] is an early exposition of the SQL-99 standard.

SQL was first defined in [4]. It was implemented as part of System R [1], one of the first generation of relational database prototypes.

There is a discussion of problems with this standard in the area of transactions and cursors in [2].

1. M. M. Astrahan et al., "System R: a relational approach to data management," *ACM Transactions on Database Systems* 1:2, pp. 97–137, 1976.
2. H. Berenson, P. A. Bernstein, J. N. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," *Proceedings of ACM SIGMOD Intl. Conf. on Management of Data*, pp. 1–10, 1995.
3. J. Celko, *SQL for Smarties*, Morgan-Kaufmann, San Francisco, 2005.
4. D. D. Chamberlin et al., "SEQUEL 2: a unified approach to data definition, manipulation, and control," *IBM Journal of Research and Development* 20:6, pp. 560–575, 1976.
5. C. J. Date and H. Darwen, *A Guide to the SQL Standard*, Addison-Wesley, Reading, MA, 1997.
6. P. Gulitzan and T. Pelzer, *SQL-99 Complete, Really*, R&D Books, Lawrence, KA, 1999.
7. J. Melton and A. R. Simon, *Understanding the New SQL: A Complete Guide*, Morgan-Kaufmann, San Francisco, 2006.



# Chapter 7

# Constraints and Triggers

In this chapter we shall cover those aspects of SQL that let us create “active” elements. An *active* element is an expression or statement that we write once and store in the database, expecting the element to execute at appropriate times. The time of action might be when a certain event occurs, such as an insertion into a particular relation, or it might be whenever the database changes so that a certain boolean-valued condition becomes true.

One of the serious problems faced by writers of applications that update the database is that the new information could be wrong in a variety of ways. For example, there are often typographical or transcription errors in manually entered data. We could write application programs in such a way that every insertion, deletion, and update command has associated with it the checks necessary to assure correctness. However, it is better to store these checks in the database, and have the DBMS administer the checks. In this way, we can be sure a check will not be forgotten, and we can avoid duplication of work.

SQL provides a variety of techniques for expressing *integrity constraints* as part of the database schema. In this chapter we shall study the principal methods. We have already seen key constraints, where an attribute or set of attributes is declared to be a key for a relation. SQL supports a form of referential integrity, called a “foreign-key constraint,” the requirement that a value in an attribute or attributes of one relation must also appear as a value in an attribute or attributes of another relation. SQL also allows constraints on attributes, constraints on tuples, and interrelation constraints called “assertions.” Finally, we discuss “triggers,” which are a form of active element that is called into play on certain specified events, such as insertion into a specific relation.

## 7.1 Keys and Foreign Keys

Recall from Section 2.3.6 that SQL allows us to define an attribute or attributes to be a key for a relation with the keywords **PRIMARY KEY** or **UNIQUE**. SQL also uses the term “key” in connection with certain referential-integrity constraints.

These constraints, called “foreign-key constraints,” assert that a value appearing in one relation must also appear in the primary-key component(s) of another relation.

### 7.1.1 Declaring Foreign-Key Constraints

A foreign key constraint is an assertion that values for certain attributes must make sense. Recall, for instance, that in Example 2.21 we considered how to express in relational algebra the constraint that the producer “certificate number” for each movie was also the certificate number of some executive in the `MovieExec` relation.

In SQL we may declare an attribute or attributes of one relation to be a *foreign key*, referencing some attribute(s) of a second relation (possibly the same relation). The implication of this declaration is twofold:

1. The referenced attribute(s) of the second relation must be declared **UNIQUE** or the **PRIMARY KEY** for their relation. Otherwise, we cannot make the foreign-key declaration.
2. Values of the foreign key appearing in the first relation must also appear in the referenced attributes of some tuple. More precisely, let there be a foreign-key  $F$  that references set of attributes  $G$  of some relation. Suppose a tuple  $t$  of the first relation has non-NULL values in all the attributes of  $F$ ; call the list of  $t$ ’s values in these attributes  $t[F]$ . Then in the referenced relation there must be some tuple  $s$  that agrees with  $t[F]$  on the attributes  $G$ . That is,  $s[G] = t[F]$ .

As for primary keys, we have two ways to declare a foreign key.

- a) If the foreign key is a single attribute we may follow its name and type by a declaration that it “references” some attribute (which must be a key — primary or unique) of some table. The form of the declaration is

**REFERENCES <table>(<attribute>)**

- b) Alternatively, we may append to the list of attributes in a **CREATE TABLE** statement one or more declarations stating that a set of attributes is a foreign key. We then give the table and its attributes (which must be a key) to which the foreign key refers. The form of this declaration is:

**FOREIGN KEY (<attributes>) REFERENCES <table>(<attributes>)**

**Example 7.1 :** Suppose we wish to declare the relation

`Studio(name, address, presC#)`

whose primary key is `name` and which has a foreign key `presC#` that references `cert#` of relation

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

We may declare `presC#` directly to reference `cert#` as follows:

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
);
```

An alternative form is to add the foreign key declaration separately, as

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT,
    FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
);
```

Notice that the referenced attribute, `cert#` in `MovieExec`, is a key of that relation, as it must be. The meaning of either of these two foreign key declarations is that whenever a value appears in the `presC#` component of a `Studio` tuple, that value must also appear in the `cert#` component of some `MovieExec` tuple. The one exception is that, should a particular `Studio` tuple have `NULL` as the value of its `presC#` component, there is no requirement that `NULL` appear as the value of a `cert#` component (but note that `cert#` is a primary key and therefore cannot have `NULL`'s anyway). □

## 7.1.2 Maintaining Referential Integrity

The schema designer may choose from among three alternatives to enforce a foreign-key constraint. We can learn the general idea by exploring Example 7.1, where it is required that a `presC#` value in relation `Studio` also be a `cert#` value in `MovieExec`. The following actions will be prevented by the DBMS (i.e., a run-time exception or error will be generated).

- a) We try to insert a new `Studio` tuple whose `presC#` value is not `NULL` and is not the `cert#` component of any `MovieExec` tuple.
- b) We try to update a `Studio` tuple to change the `presC#` component to a non-`NULL` value that is not the `cert#` component of any `MovieExec` tuple.
- c) We try to delete a `MovieExec` tuple, and its `cert#` component, which is not `NULL`, appears as the `presC#` component of one or more `Studio` tuples.

- d) We try to update a `MovieExec` tuple in a way that changes the `cert#` value, and the old `cert#` is the value of `presC#` of some movie studio.

For the first two modifications, where the change is to the relation where the foreign-key constraint is declared, there is no alternative; the system has to reject the violating modification. However, for changes to the referenced relation, of which the last two modifications are examples, the designer can choose among three options:

1. *The Default Policy: Reject Violating Modifications.* SQL has a default policy that any modification violating the referential integrity constraint is rejected.
2. *The Cascade Policy.* Under this policy, changes to the referenced attribute(s) are mimicked at the foreign key. For example, under the cascade policy, when we delete the `MovieExec` tuple for the president of a studio, then to maintain referential integrity the system will delete the referencing tuple(s) from `Studio`. If we update the `cert#` for some movie executive from  $c_1$  to  $c_2$ , and there was some `Studio` tuple with  $c_1$  as the value of its `presC#` component, then the system will also update this `presC#` component to have value  $c_2$ .
3. *The Set-Null Policy.* Here, when a modification to the referenced relation affects a foreign-key value, the latter is changed to `NULL`. For instance, if we delete from `MovieExec` the tuple for a president of a studio, the system would change the `presC#` value for that studio to `NULL`. If we updated that president's certificate number in `MovieExec`, we would again set `presC#` to `NULL` in `Studio`.

These options may be chosen for deletes and updates, independently, and they are stated with the declaration of the foreign key. We declare them with `ON DELETE` or `ON UPDATE` followed by our choice of `SET NULL` or `CASCADE`.

**Example 7.2:** Let us see how we might modify the declaration of

```
Studio(name, address, presC#)
```

in Example 7.1 to specify the handling of deletes and updates in the

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

relation. Figure 7.1 takes the first of the `CREATE TABLE` statements in that example and expands it with `ON DELETE` and `ON UPDATE` clauses. Line (5) says that when we delete a `MovieExec` tuple, we set the `presC#` of any studio of which he or she was the president to `NULL`. Line (6) says that if we update the `cert#` component of a `MovieExec` tuple, then any tuples in `Studio` with the same value in the `presC#` component are changed similarly.

```

1) CREATE TABLE Studio (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     presC# INT REFERENCES MovieExec(cert#)
5)         ON DELETE SET NULL
6)         ON UPDATE CASCADE
);

```

Figure 7.1: Choosing policies to preserve referential integrity

## Dangling Tuples and Modification Policies

A tuple with a foreign key value that does not appear in the referenced relation is said to be a *dangling tuple*. Recall that a tuple which fails to participate in a join is also called “dangling.” The two ideas are closely related. If a tuple’s foreign-key value is missing from the referenced relation, then the tuple will not participate in a join of its relation with the referenced relation, if the join is on equality of the foreign key and the key it references (called a *foreign-key join*). The dangling tuples are exactly the tuples that violate referential integrity for this foreign-key constraint.

Note that in this example, the set-null policy makes more sense for deletes, while the cascade policy seems preferable for updates. We would expect that if, for instance, a studio president retires, the studio will exist with a “null” president for a while. However, an update to the certificate number of a studio president is most likely a clerical change. The person continues to exist and to be the president of the studio, so we would like the `presC#` attribute in `Studio` to follow the change. □

### 7.1.3 Deferred Checking of Constraints

Let us assume the situation of Example 7.1, where `presC#` in `Studio` is a foreign key referencing `cert#` of `MovieExec`. Arnold Schwarzenegger retires as Governor of California and decides to found a movie studio, called La Vista Studios, of which he will naturally be the president. If we execute the insertion:

```

INSERT INTO Studio
VALUES('La Vista', 'New York', 23456);

```

we are in trouble. The reason is that there is no tuple of `MovieExec` with certificate number 23456 (the presumed newly issued certificate for Arnold Schwarzenegger), so there is an obvious violation of the foreign-key constraint.

One possible fix is first to insert the tuple for La Vista without a president's certificate, as:

```
INSERT INTO Studio(name, address)
VALUES('La Vista', 'New York');
```

This change avoids the constraint violation, because the La-Vista tuple is inserted with NULL as the value of `presC#`, and NULL in a foreign key does not require that we check for the existence of any value in the referenced column. However, we must insert a tuple for Arnold Schwarzenegger into `MovieExec`, with his correct certificate number before we can apply an update statement such as

```
UPDATE Studio
SET presC# = 23456
WHERE name = 'La Vista';
```

If we do not fix `MovieExec` first, then this update statement will also violate the foreign-key constraint.

Of course, inserting Arnold Schwarzenegger and his certificate number into `MovieExec` before inserting La Vista into `Studio` will surely protect against a foreign-key violation in this case. However, there are cases of *circular constraints* that cannot be fixed by judiciously ordering the database modification steps we take.

**Example 7.3:** If movie executives were limited to studio presidents, then we might want to declare `cert#` to be a foreign key referencing `Studio(presC#)`; we would first have to declare `presC#` to be `UNIQUE`, but that declaration makes sense if you assume a person cannot be the president of two studios at the same time.

Now, it is impossible to insert new studios with new presidents. We can't insert a tuple with a new value of `presC#` into `Studio`, because that tuple would violate the foreign-key constraint from `presC#` to `MovieExec(cert#)`. We can't insert a tuple with a new value of `cert#` into `MovieExec`, because that would violate the foreign-key constraint from `cert#` to `Studio(presC#)`. □

The problem of Example 7.3 can be solved as follows.

1. First, we must group the two insertions (one into `Studio` and the other into `MovieExec`) into a single transaction.
2. Then, we need a way to tell the DBMS not to check the constraints until after the whole transaction has finished its actions and is about to commit.

To inform the DBMS about point (2), the declaration of any constraint — key, foreign-key, or other constraint types we shall meet later in this chapter — may be followed by one of `DEFERRABLE` or `NOT DEFERRABLE`. The latter is the

default, and means that every time a database modification statement is executed, the constraint is checked immediately afterwards, if the modification could violate the foreign-key constraint. However, if we declare a constraint to be **DEFERRABLE**, then we have the option of having it wait until a transaction is complete before checking the constraint.

We follow the keyword **DEFERRABLE** by either **INITIALLY DEFERRED** or **INITIALLY IMMEDIATE**. In the former case, checking will be deferred to just before each transaction commits. In the latter case, the check will be made immediately after each statement.

**Example 7.4:** Figure 7.2 shows the declaration of **Studio** modified to allow the checking of its foreign-key constraint to be deferred until the end of each transaction. We have also declared **presC#** to be **UNIQUE**, in order that it may be referenced by other relations' foreign-key constraints.

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT UNIQUE
        REFERENCES MovieExec(cert#)
        DEFERRABLE INITIALLY DEFERRED
);
```

Figure 7.2: Making **presC#** unique and deferring the checking of its foreign-key constraint

If we made a similar declaration for the hypothetical foreign-key constraint from **MovieExec(cert#)** to **Studio(presC#)** mentioned in Example 7.3, then we could write transactions that inserted two tuples, one into each relation, and the two foreign-key constraints would not be checked until after both insertions had been done. Then, if we insert both a new studio and its new president, and use the same certificate number in each tuple, we would avoid violation of any constraint. □

There are two additional points about deferring constraints that we should bear in mind:

- Constraints of any type can be given names. We shall discuss how to do so in Section 7.3.1.
- If a constraint has a name, say **MyConstraint**, then we can change a deferrable constraint from immediate to deferred by the SQL statement

```
SET CONSTRAINT MyConstraint DEFERRED;
```

and we can reverse the process by replacing **DEFERRED** in the above to **IMMEDIATE**.

### 7.1.4 Exercises for Section 7.1

**Exercise 7.1.1:** Our running example movie database of Section 2.2.8 has keys defined for all its relations.

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

Declare the following referential integrity constraints for the movie database as in Exercise 7.1.1.

- The producer of a movie must be someone mentioned in MovieExec. Modifications to MovieExec that violate this constraint are rejected.
- Repeat (a), but violations result in the producerC# in Movie being set to NULL.
- Repeat (a), but violations result in the deletion or update of the offending Movie tuple.
- A movie that appears in StarsIn must also appear in Movie. Handle violations by rejecting the modification.
- A star appearing in StarsIn must also appear in MovieStar. Handle violations by deleting violating tuples.

**Exercise 7.1.2:** We would like to declare the constraint that every movie in the relation Movie must appear with at least one star in StarsIn. Can we do so with a foreign-key constraint? Why or why not?

**Exercise 7.1.3:** Suggest suitable keys and foreign keys for the relations of the PC database:

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

of Exercise 2.4.1. Modify your SQL schema from Exercise 2.3.1 to include declarations of these keys.

**Exercise 7.1.4:** Suggest suitable keys for the relations of the battleships database

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

of Exercise 2.4.3. Modify your SQL schema from Exercise 2.3.2 to include declarations of these keys.

**Exercise 7.1.5:** Write the following referential integrity constraints for the battleships database as in Exercise 7.1.4. Use your assumptions about keys from that exercise, and handle all violations by setting the referencing attribute value to `NULL`.

- a) Every class mentioned in `Ships` must be mentioned in `Classes`.
- b) Every battle mentioned in `Outcomes` must be mentioned in `Battles`.
- c) Every ship mentioned in `Outcomes` must be mentioned in `Ships`.

## 7.2 Constraints on Attributes and Tuples

Within a SQL `CREATE TABLE` statement, we can declare two kinds of constraints:

1. A constraint on a single attribute.
2. A constraint on a tuple as a whole.

In Section 7.2.1 we shall introduce a simple type of constraint on an attribute's value: the constraint that the attribute not have a `NULL` value. Then in Section 7.2.2 we cover the principal form of constraints of type (1): *attribute-based CHECK constraints*. The second type, the tuple-based constraints, are covered in Section 7.2.3.

There are other, more general kinds of constraints that we shall meet in Sections 7.4 and 7.5. These constraints can be used to restrict changes to whole relations or even several relations, as well as to constrain the value of a single attribute or tuple.

### 7.2.1 Not-Null Constraints

One simple constraint to associate with an attribute is `NOT NULL`. The effect is to disallow tuples in which this attribute is `NULL`. The constraint is declared by the keywords `NOT NULL` following the declaration of the attribute in a `CREATE TABLE` statement.

**Example 7.5:** Suppose relation `Studio` required `presC#` not to be `NULL`, perhaps by changing line (4) of Fig. 7.1 to:

4) `presC# INT REFERENCES MovieExec(cert#) NOT NULL`

This change has several consequences. For instance:

- We could not insert a tuple into `Studio` by specifying only the name and address, because the inserted tuple would have `NULL` in the `presC#` component.
- We could not use the set-null policy in situations like line (5) of Fig. 7.1, which tells the system to fix foreign-key violations by making `presC#` be `NULL`.

□

## 7.2.2 Attribute-Based CHECK Constraints

More complex constraints can be attached to an attribute declaration by the keyword `CHECK` and a parenthesized condition that must hold for every value of this attribute. In practice, an attribute-based `CHECK` constraint is likely to be a simple limit on values, such as an enumeration of legal values or an arithmetic inequality. However, in principle the condition can be anything that could follow `WHERE` in a SQL query. This condition may refer to the attribute being constrained, by using the name of that attribute in its expression. However, if the condition refers to any other relations or attributes of relations, then the relation must be introduced in the `FROM` clause of a subquery (even if the relation referred to is the one to which the checked attribute belongs).

An attribute-based `CHECK` constraint is checked whenever any tuple gets a new value for this attribute. The new value could be introduced by an update for the tuple, or it could be part of an inserted tuple. In the case of an update, the constraint is checked on the new value, not the old value. If the constraint is violated by the new value, then the modification is rejected.

It is important to understand that an attribute-based `CHECK` constraint is not checked if the database modification does not change the attribute with which the constraint is associated. This limitation can result in the constraint becoming violated, if other values involved in the constraint do change. First, let us consider a simple example of an attribute-based check. Then we shall see a constraint that involves a subquery, and also see the consequence of the fact that the constraint is only checked when its attribute is modified.

**Example 7.6:** Suppose we want to require that certificate numbers be at least six digits. We could modify line (4) of Fig. 7.1, a declaration of the schema for relation

`Studio(name, address, presC#)`

to be

```
4)    presC# INT REFERENCES MovieExec(cert#)
          CHECK (presC# >= 100000)
```

For another example, the attribute `gender` of relation

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

was declared in Fig. 2.8 to be of data type CHAR(1) — that is, a single character. However, we really expect that the only characters that will appear there are 'F' and 'M'. The following substitute for line (4) of Fig. 2.8 enforces the rule:

```
4) gender CHAR(1) CHECK (gender IN ('F', 'M')),
```

Note that the expression ('F' 'M') describes a one-component relation with two tuples. The constraint says that the value of any **gender** component must be in this set. □

**Example 7.7:** We might suppose that we could simulate a referential integrity constraint by an attribute-based CHECK constraint that requires the existence of the referred-to value. The following is an *erroneous* attempt to simulate the requirement that the **presC#** value in a

```
Studio(name, address, presC#)
```

tuple must appear in the **cert#** component of some

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

tuple. Suppose line (4) of Fig. 7.1 were replaced by

```
4) presC# INT CHECK  
(presC# IN (SELECT cert# FROM MovieExec))
```

This statement is a legal attribute-based CHECK constraint, but let us look at its effect. Modifications to **Studio** that introduce a **presC#** that is not also a **cert#** of **MovieExec** will be rejected. That is almost what the similar foreign-key constraint would do, except that the attribute-based check will also reject a NULL value for **presC#** if there is no NULL value for **cert#**. But far more importantly, if we change the **MovieExec** relation, say by deleting the tuple for the president of a studio, this change is invisible to the above CHECK constraint. Thus, the deletion is permitted, even though the attribute-based CHECK constraint on **presC#** is now violated. □

### 7.2.3 Tuple-Based CHECK Constraints

To declare a constraint on the tuples of a single table *R*, we may add to the list of attributes and key or foreign-key declarations, in *R*'s CREATE TABLE statement, the keyword CHECK followed by a parenthesized condition. This condition can be anything that could appear in a WHERE clause. It is interpreted as a condition about a tuple in the table *R*, and the attributes of *R* may be referred to by name in this expression. However, as for attribute-based CHECK constraints, the condition may also mention, in subqueries, other relations or other tuples of the same relation *R*.

## Limited Constraint Checking: Bug or Feature?

One might wonder why attribute- and tuple-based checks are allowed to be violated if they refer to other relations or other tuples of the same relation. The reason is that such constraints can be implemented much more efficiently than more general constraints can. With attribute- or tuple-based checks, we only have to evaluate that constraint for the tuple(s) that are inserted or updated. On the other hand, assertions must be evaluated every time any one of the relations they mention is changed. The careful database designer will use attribute- and tuple-based checks only when there is no possibility that they will be violated, and will use another mechanism, such as assertions (Section 7.4) or triggers (Section 7.5) otherwise.

The condition of a tuple-based **CHECK** constraint is checked every time a tuple is inserted into  $R$  and every time a tuple of  $R$  is updated. The condition is evaluated for the new or updated tuple. If the condition is false for that tuple, then the constraint is violated and the insertion or update statement that caused the violation is rejected. However, if the condition mentions some other relation in a subquery, and a change to that relation causes the condition to become false for some tuple of  $R$ , the check does not inhibit this change. That is, like an attribute-based **CHECK**, a tuple-based **CHECK** is invisible to other relations. In fact, even a deletion from  $R$  can cause the condition to become false, if  $R$  is mentioned in a subquery.

On the other hand, if a tuple-based check does not have subqueries, then we can rely on its always holding. Here is an example of a tuple-based **CHECK** constraint that involves several attributes of one tuple, but no subqueries.

**Example 7.8:** Recall Example 2.3, where we declared the schema of table `MovieStar`. Figure 7.3 repeats the **CREATE TABLE** statement with the addition of a primary-key declaration and one other constraint, which is one of several possible “consistency conditions” that we might wish to check. This constraint says that if the star’s gender is male, then his name must not begin with ‘Ms.’.

In line (2), `name` is declared the primary key for the relation. Then line (6) declares a constraint. The condition of this constraint is true for every female movie star and for every star whose name does not begin with ‘Ms.’. The only tuples for which it is *not* true are those where the gender is male and the name *does* begin with ‘Ms.’. Those are exactly the tuples we wish to exclude from `MovieStar`.  $\square$

```

1) CREATE TABLE MovieStar (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     gender CHAR(1),
5)     birthdate DATE,
6)     CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
);

```

Figure 7.3: A constraint on the table `MovieStar`

### Writing Constraints Correctly

Many constraints are like Example 7.8, where we want to forbid tuples that satisfy two or more conditions. The expression that should follow the check is the `OR` of the negations, or opposites, of each condition; this transformation is one of “DeMorgan’s laws”: the negation of the `AND` of terms is the `OR` of the negations of the same terms. Thus, in Example 7.8 the first condition was that the star is male, and we used `gender = 'F'` as a suitable negation (although perhaps `gender <> 'M'` would be the more normal way to phrase the negation). The second condition is that the `name` begins with '`Ms.`', and for this negation we used the `NOT LIKE` comparison. This comparison negates the condition itself, which would be `name LIKE 'Ms.%'` in SQL.

#### 7.2.4 Comparison of Tuple- and Attribute-Based Constraints

If a constraint on a tuple involves more than one attribute of that tuple, then it must be written as a tuple-based constraint. However, if the constraint involves only one attribute of the tuple, then it can be written as either a tuple- or attribute-based constraint. In either case, we do not count attributes mentioned in subqueries, so even a attribute-based constraint can mention other attributes of the same relation in subqueries.

When only one attribute of the tuple is involved (not counting subqueries), then the condition checked is the same, regardless of whether a tuple- or attribute-based constraint is written. However, the tuple-based constraint will be checked more frequently than the attribute-based constraint — whenever any attribute of the tuple changes, rather than only when the attribute mentioned in the constraint changes.

#### 7.2.5 Exercises for Section 7.2

**Exercise 7.2.1:** Write the following constraints for attributes of the relation

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

- a) The year cannot be before 1915.
- b) The length cannot be less than 60 nor more than 250.
- c) The studio name can only be Disney, Fox, MGM, or Paramount.

**Exercise 7.2.2:** Write the following constraints on attributes from our example schema

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

of Exercise 2.4.1.

- a) The speed of a laptop must be at least 2.0.
- b) The only types of printers are laser, ink-jet, and bubble-jet.
- c) The only types of products are PC's, laptops, and printers.
- ! d) A model of a product must also be the model of a PC, a laptop, or a printer.

**Exercise 7.2.3:** Write the following constraints as tuple-based CHECK constraints on one of the relations of our running movies example:

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

If the constraint actually involves two relations, then you should put constraints in both relations so that whichever relation changes, the constraint will be checked on insertions and updates. Assume no deletions; it is not always possible to maintain tuple-based constraints in the face of deletions.

- a) A star may not appear in a movie made before they were born.
- ! b) No two studios may have the same address.
- ! c) A name that appears in `MovieStar` must not also appear in `MovieExec`.
- ! d) A studio name that appears in `Studio` must also appear in at least one `Movies` tuple.

- !! e) If a producer of a movie is also the president of a studio, then they must be the president of the studio that made the movie.

**Exercise 7.2.4:** Write the following as tuple-based CHECK constraints about our “PC” schema.

- a) A PC with a processor speed less than 2.0 must not sell for more than \$600.
- b) A laptop with a screen size less than 15 inches must have at least a 40 gigabyte hard disk or sell for less than \$1000.

**Exercise 7.2.5:** Write the following as tuple-based CHECK constraints about our “battleships” schema:

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

- a) No class of ships may have guns with larger than a 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 ship can be in battle before it is launched.

**Exercise 7.2.6:** In Examples 7.6 and 7.8, we introduced constraints on the `gender` attribute of `MovieStar`. What restrictions, if any, do each of these constraints enforce if the value of `gender` is `NULL`?

## 7.3 Modification of Constraints

It is possible to add, modify, or delete constraints at any time. The way to express such modifications depends on whether the constraint involved is associated with an attribute, a table, or (as in Section 7.4) a database schema.

### 7.3.1 Giving Names to Constraints

In order to modify or delete an existing constraint, it is necessary that the constraint have a name. To do so, we precede the constraint by the keyword `CONSTRAINT` and a name for the constraint.

**Example 7.9:** We could rewrite line (2) of Fig. 2.9 to name the constraint that says attribute `name` is a primary key, as

```
2)      name CHAR(30) CONSTRAINT NameIsKey PRIMARY KEY,
```

Similarly, we could name the attribute-based CHECK constraint that appeared in Example 7.6 by:

```
4) gender CHAR(1) CONSTRAINT NoAndro
    CHECK (gender IN ('F', 'M')),
```

Finally, the following constraint:

```
6)      CONSTRAINT RightTitle
        CHECK (gender = 'F' OR name NOT LIKE 'Ms.%');
```

is a rewriting of the tuple-based CHECK constraint in line (6) of Fig. 7.3 to give that constraint a name. □

### 7.3.2 Altering Constraints on Tables

We mentioned in Section 7.1.3 that we can switch the checking of a constraint from immediate to deferred or vice-versa with a **SET CONSTRAINT** statement. Other changes to constraints are effected with an **ALTER TABLE** statement. We previously discussed some uses of the **ALTER TABLE** statement in Section 2.3.4, where we used it to add and delete attributes.

**ALTER TABLE** statements can affect constraints in several ways. You may drop a constraint with keyword **DROP** and the name of the constraint to be dropped. You may also add a constraint with the keyword **ADD**, followed by the constraint to be added. Note, however, that the added constraint must be of a kind that can be associated with tuples, such as tuple-based constraints, key, or foreign-key constraints. Also note that you cannot add a constraint to a table unless it holds at that time for every tuple in the table.

**Example 7.10:** Let us see how we would drop and add the constraints of Example 7.9 on relation **MovieStar**. The following sequence of three statements drops them:

```
ALTER TABLE MovieStar DROP CONSTRAINT NameIsKey;
ALTER TABLE MovieStar DROP CONSTRAINT NoAndro;
ALTER TABLE MovieStar DROP CONSTRAINT RightTitle;
```

Should we wish to reinstate these constraints, we would alter the schema for relation **MovieStar** by adding the same constraints, for example:

```
ALTER TABLE MovieStar ADD CONSTRAINT NameIsKey
    PRIMARY KEY (name);
ALTER TABLE MovieStar ADD CONSTRAINT NoAndro
    CHECK (gender IN ('F', 'M'));
ALTER TABLE MovieStar ADD CONSTRAINT RightTitle
    CHECK (gender = 'F' OR name NOT LIKE 'Ms.%');
```

## Name Your Constraints

Remember, it is a good idea to give each of your constraints a name, even if you do not believe you will ever need to refer to it. Once the constraint is created without a name, it is too late to give it one later, should you wish to alter it. However, should you be faced with a situation of having to alter a nameless constraint, you will find that your DBMS probably has a way for you to query it for a list of all your constraints, and that it has given your unnamed constraint an internal name of its own, which you may use to refer to the constraint.

These constraints are now tuple-based, rather than attribute-based checks. We cannot bring them back as attribute-based constraints.

The name is optional for these reintroduced constraints. However, we cannot rely on SQL remembering the dropped constraints. Thus, when we add a former constraint we need to write the constraint again; we cannot refer to it by its former name. □

### 7.3.3 Exercises for Section 7.3

**Exercise 7.3.1:** Show how to alter your relation schemas for the movie example:

```
Movie(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

in the following ways.

- a) Make `title` and `year` the key for `Movie`.
- b) Require the referential integrity constraint that the producer of every movie appear in `MovieExec`.
- c) Require that no movie length be less than 60 nor greater than 250.
- d) Require that no name appear as both a movie star and movie executive (this constraint need not be maintained in the face of deletions).
- e) Require that no two studios have the same address.

**Exercise 7.3.2:** Show how to alter the schemas of the “battleships” database:

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

to have the following tuple-based constraints.

- a) Class and country form a key for relation **Classes**.
- b) Require the referential integrity constraint that every ship appearing in **Battles** also appears in **Ships**.
- c) Require the referential integrity constraint that every ship appearing in **Outcomes** appears in **Ships**.
- d) Require that no ship has more than 14 guns.
- ! e) Disallow a ship being in battle before it is launched.

## 7.4 Assertions .

The most powerful forms of active elements in SQL are not associated with particular tuples or components of tuples. These elements, called “triggers” and “assertions,” are part of the database schema, on a par with tables.

- An assertion is a boolean-valued SQL expression that must be true at all times.
- A trigger is a series of actions that are associated with certain events, such as insertions into a particular relation, and that are performed whenever these events arise.

Assertions are easier for the programmer to use, since they merely require the programmer to state what must be true. However, triggers are the feature DBMS’s typically provide as general-purpose, active elements. The reason is that it is very hard to implement assertions efficiently. The DBMS must deduce whether any given database modification could affect the truth of an assertion. Triggers, on the other hand, tell exactly when the DBMS needs to deal with them.

### 7.4.1 Creating Assertions

The SQL standard proposes a simple form of *assertion* that allows us to enforce any condition (expression that can follow WHERE). Like other schema elements, we declare an assertion with a CREATE statement. The form of an assertion is:

```
CREATE ASSERTION <assertion-name> CHECK (<condition>)
```

The condition in an assertion must be true when the assertion is created and must remain true; any database modification that causes it to become false will be rejected.<sup>1</sup> Recall that the other types of CHECK constraints we have covered can be violated under certain conditions, if they involve subqueries.

### 7.4.2 Using Assertions

There is a difference between the way we write tuple-based CHECK constraints and the way we write assertions. Tuple-based checks can refer directly to the attributes of that relation in whose declaration they appear. An assertion has no such privilege. Any attributes referred to in the condition must be introduced in the assertion, typically by mentioning their relation in a select-from-where expression.

Since the condition must have a boolean value, it is necessary to combine results in some way to make a single true/false choice. For example, we might write the condition as an expression producing a relation, to which NOT EXISTS is applied; that is, the constraint is that this relation is always empty. Alternatively, we might apply an aggregation operator like SUM to a column of a relation and compare it to a constant. For instance, this way we could require that a sum always be less than some limiting value.

**Example 7.11:** Suppose we wish to require that no one can become the president of a studio unless their net worth is at least \$10,000,000. We declare an assertion to the effect that the set of movie studios with presidents having a net worth less than \$10,000,000 is empty. This assertion involves the two relations

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

The assertion is shown in Fig. 7.4. □

```
CREATE ASSERTION RichPres CHECK
  (NOT EXISTS
    (SELECT Studio.name
     FROM Studio, MovieExec
     WHERE presC# = cert# AND netWorth < 10000000
    )
  );
```

Figure 7.4: Assertion guaranteeing rich studio presidents

---

<sup>1</sup> However, remember from Section 7.1.3 that it is possible to defer the checking of a constraint until just before its transaction commits. If we do so with an assertion, it may briefly become false until the end of a transaction.

**Example 7.12:** Here is another example of an assertion. It involves the relation

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

and says the total length of all movies by a given studio shall not exceed 10,000 minutes.

```
CREATE ASSERTION SumLength CHECK (10000 >= ALL
    (SELECT SUM(length) FROM Movies GROUP BY studioName)
);
```

As this constraint involves only the relation `Movies`, it seemingly could have been expressed as a tuple-based `CHECK` constraint in the schema for `Movies` rather than as an assertion. That is, we could add to the definition of table `Movies` the tuple-based `CHECK` constraint

```
CHECK (10000 >= ALL
    (SELECT SUM(length) FROM Movies GROUP BY studioName));
```

Notice that in principle this condition applies to every tuple of table `Movies`. However, it does not mention any attributes of the tuple explicitly, and all the work is done in the subquery.

Also observe that if implemented as a tuple-based constraint, the check would not be made on deletion of a tuple from the relation `Movies`. In this example, that difference causes no harm, since if the constraint was satisfied before the deletion, then it is surely satisfied after the deletion. However, if the constraint were a lower bound on total length, rather than an upper bound as in this example, then we could find the constraint violated had we written it as a tuple-based check rather than an assertion. □

As a final point, it is possible to drop an assertion. The statement to do so follows the pattern for any database schema element:

```
DROP ASSERTION <assertion name>
```

### 7.4.3 Exercises for Section 7.4

**Exercise 7.4.1:** Write the following assertions. The database schema is from the “PC” example of Exercise 2.4.1:

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

- a) No manufacturer of PC's may also make laptops.

## Comparison of Constraints

The following table lists the principal differences among attribute-based checks, tuple-based checks, and assertions.

Type of Constraint	Where Declared	When Activated	Guaranteed to Hold?
Attribute-based CHECK	With attribute	On insertion to relation or attribute update	Not if subqueries
Tuple-based CHECK	Element of relation schema	On insertion to relation or tuple update	Not if subqueries
Assertion	Element of database schema	On any change to any mentioned relation	Yes

- b) A manufacturer of a PC must also make a laptop with at least as great a processor speed.
- c) If a laptop has a larger main memory than a PC, then the laptop must also have a higher price than the PC.
- d) If the relation **Product** mentions a model and its type, then this model must appear in the relation appropriate to that type.

**Exercise 7.4.2:** Write the following as assertions. The database schema is from the battleships example of Exercise 2.4.3.

```

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

```

- a) No class may have more than 2 ships.
- ! b) No country may have both battleships and battlecruisers.
- ! c) No ship with more than 9 guns may be in a battle with a ship having fewer than 9 guns that was sunk.
- ! d) No ship may be launched before the ship that bears the name of the first ship's class.
- ! e) For every class, there is a ship with the name of that class.

**! Exercise 7.4.3:** The assertion of Exercise 7.11 can be written as two tuple-based constraints. Show how to do so.

## 7.5 Triggers

*Triggers*, sometimes called *event-condition-action rules* or *ECA rules*, differ from the kinds of constraints discussed previously in three ways.

1. Triggers are only awakened when certain *events*, specified by the database programmer, occur. The sorts of events allowed are usually insert, delete, or update to a particular relation. Another kind of event allowed in many SQL systems is a transaction end.
2. Once awakened by its triggering event, the trigger tests a *condition*. If the condition does not hold, then nothing else associated with the trigger happens in response to this event.
3. If the condition of the trigger is satisfied, the *action* associated with the trigger is performed by the DBMS. A possible action is to modify the effects of the event in some way, even aborting the transaction of which the event is part. However, the action could be any sequence of database operations, including operations not connected in any way to the triggering event.

### 7.5.1 Triggers in SQL

The SQL trigger statement gives the user a number of different options in the event, condition, and action parts. Here are the principal features.

1. The check of the trigger's condition and the action of the trigger may be executed either on the *state of the database* (i.e., the current instances of all the relations) that exists before the triggering event is itself executed or on the state that exists after the triggering event is executed.
2. The condition and action can refer to both old and/or new values of tuples that were updated in the triggering event.
3. It is possible to define update events that are limited to a particular attribute or set of attributes.
4. The programmer has an option of specifying that the trigger executes either:
  - (a) Once for each modified tuple (a *row-level trigger*), or
  - (b) Once for all the tuples that are changed in one SQL statement (a *statement-level trigger*; remember that one SQL modification statement can affect many tuples).

Before giving the details of the syntax for triggers, let us consider an example that will illustrate the most important syntactic as well as semantic points. Notice in the example trigger, Fig. 7.5, the key elements and the order in which they appear:

- a) The CREATE TRIGGER statement (line 1).
- b) A clause indicating the triggering event and telling whether the trigger uses the database state before or after the triggering event (line 2).
- c) A REFERENCING clause to allow the condition and action of the trigger to refer to the tuple being modified (lines 3 through 5). In the case of an update, such as this one, this clause allows us to give names to the tuple both before and after the change.
- d) A clause telling whether the trigger executes once for each modified row or once for all the modifications made by one SQL statement (line 6).
- e) The condition, which uses the keyword WHEN and a boolean expression (line 7).
- f) The action, consisting of one or more SQL statements (lines 8 through 10).

Each of these elements has options, which we shall discuss after working through the example.

**Example 7.13:** In Fig. 7.13 is a SQL trigger that applies to the

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

table. It is triggered by updates to the netWorth attribute. The effect of this trigger is to foil any attempt to lower the net worth of a movie executive.

```
1) CREATE TRIGGER NetWorthTrigger
2) AFTER UPDATE OF netWorth ON MovieExec
3) REFERENCING
4)     OLD ROW AS OldTuple,
5)     NEW ROW AS NewTuple
6) FOR EACH ROW
7) WHEN (OldTuple.netWorth > NewTuple.netWorth)
8)     UPDATE MovieExec
9)     SET netWorth = OldTuple.netWorth
10)    WHERE cert# = NewTuple.cert#;
```

Figure 7.5: A SQL trigger

Line (1) introduces the declaration with the keywords `CREATE TRIGGER` and the name of the trigger. Line (2) then gives the triggering event, namely the update of the `netWorth` attribute of the `MovieExec` relation. Lines (3) through (5) set up a way for the condition and action portions of this trigger to talk about both the old tuple (the tuple before the update) and the new tuple (the tuple after the update). These tuples will be referred to as `OldTuple` and `NewTuple`, according to the declarations in lines (4) and (5), respectively. In the condition and action, these names can be used as if they were tuple variables declared in the `FROM` clause of an ordinary SQL query.

Line (6), the phrase `FOR EACH ROW`, expresses the requirement that this trigger is executed once for each updated tuple. Line (7) is the condition part of the trigger. It says that we only perform the action when the new net worth is lower than the old net worth; i.e., the net worth of an executive has shrunk.

Lines (8) through (10) form the action portion. This action is an ordinary SQL update statement that has the effect of restoring the net worth of the executive to what it was before the update. Note that in principle, every tuple of `MovieExec` is considered for update, but the `WHERE`-clause of line (10) guarantees that only the updated tuple (the one with the proper `cert#`) will be affected.

□

## 7.5.2 The Options for Trigger Design

Of course Example 7.13 illustrates only some of the features of SQL triggers. In the points that follow, we shall outline the options that are offered by triggers and how to express these options.

- Line (2) of Fig. 7.5 says that the condition test and action of the rule are executed on the database state that exists after the triggering event, as indicated by the keyword `AFTER`. We may replace `AFTER` by `BEFORE`, in which case the `WHEN` condition is tested on the database state that exists before the triggering event is executed. If the condition is true, then the action of the trigger is executed on that state. Finally, the event that awakened the trigger is executed, regardless of whether the condition is still true. There is a third option, `INSTEAD OF`, that we discuss in Section 8.2.3, in connection with modification of views.
- Besides `UPDATE`, other possible triggering events are `INSERT` and `DELETE`. The `OF` `netWorth` clause in line (2) of Fig. 7.5 is optional for `UPDATE` events, and if present defines the event to be only an update of the attribute(s) listed after the keyword `OF`. An `OF` clause is not permitted for `INSERT` or `DELETE` events; these events make sense for entire tuples only.
- The `WHEN` clause is optional. If it is missing, then the action is executed whenever the trigger is awakened. If present, then the action is executed only if the condition following `WHEN` is true.

- While we showed a single SQL statement as an action, there can be any number of such statements, separated by semicolons and surrounded by `BEGIN...END`.
- When the triggering event of a row-level trigger is an update, then there will be old and new tuples, which are the tuple before the update and after, respectively. We give these tuples names by the `OLD ROW AS` and `NEW ROW AS` clauses seen in lines (4) and (5). If the triggering event is an insertion, then we may use a `NEW ROW AS` clause to give a name for the inserted tuple, and `OLD ROW AS` is disallowed. Conversely, on a deletion `OLD ROW AS` is used to name the deleted tuple and `NEW ROW AS` is disallowed.
- If we omit the `FOR EACH ROW` on line (6) or replace it by the default `FOR EACH STATEMENT`, then a row-level trigger such as Fig. 7.5 becomes a statement-level trigger. A statement-level trigger is executed once whenever a statement of the appropriate type is executed, no matter how many rows — zero, one, or many — it actually affects. For instance, if we update an entire table with a SQL update statement, a statement-level update trigger would execute only once, while a row-level trigger would execute once for each tuple to which an update was applied.
- In a statement-level trigger, we cannot refer to old and new tuples directly, as we did in lines (4) and (5). However, any trigger — whether row- or statement-level — can refer to the relation of *old tuples* (deleted tuples or old versions of updated tuples) and the relation of *new tuples* (inserted tuples or new versions of updated tuples), using declarations such as `OLD TABLE AS OldStuff` and `NEW TABLE AS NewStuff`.

**Example 7.14:** Suppose we want to prevent the average net worth of movie executives from dropping below \$500,000. This constraint could be violated by an insertion, a deletion, or an update to the `netWorth` column of

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

The subtle point is that we might, in one statement insert, delete, or change many tuples of `MovieExec`. During the modification, the average net worth might temporarily dip below \$500,000 and then rise above it by the time all the modifications are made. We only want to reject the entire set of modifications if the net worth is below \$500,000 at the end of the statement.

It is necessary to write one trigger for each of these three events: insert, delete, and update of relation `MovieExec`. Figure 7.6 shows the trigger for the update event. The triggers for the insertion and deletion of tuples are similar.

Lines (3) through (5) declare that `NewStuff` and `OldStuff` are the names of relations containing the new tuples and old tuples that are involved in the database operation that awakened our trigger. Note that one database statement can modify many tuples of a relation, and if such a statement executes, there can be many tuples in `NewStuff` and `OldStuff`.

```

1) CREATE TRIGGER AvgNetWorthTrigger
2) AFTER UPDATE OF netWorth ON MovieExec
3) REFERENCING
4)     OLD TABLE AS OldStuff,
5)     NEW TABLE AS NewStuff
6) FOR EACH STATEMENT
7) WHEN (500000 > (SELECT AVG(netWorth) FROM MovieExec))
8) BEGIN
9)     DELETE FROM MovieExec
10)    WHERE (name, address, cert#, netWorth) IN NewStuff;
11)    INSERT INTO MovieExec
12)        (SELECT * FROM OldStuff);
13) END;

```

Figure 7.6: Constraining the average net worth

If the operation is an update, then tables **NewStuff** and **OldStuff** are the new and old versions of the updated tuples, respectively. If an analogous trigger were written for deletions, then the deleted tuples would be in **OldStuff**, and there would be no declaration of a relation name like **NewStuff** for **NEW TABLE** in this trigger. Likewise, in the analogous trigger for insertions, the new tuples would be in **NewStuff**, and there would be no declaration of **OldStuff**.

Line (6) tells us that this trigger is executed once for a statement, regardless of how many tuples are modified. Line (7) is the condition. This condition is satisfied if the average net worth *after* the update is less than \$500,000.

The action of lines (8) through (13) consists of two statements that restore the old relation **MovieExec** if the condition of the **WHEN** clause is satisfied; i.e., the new average net worth is too low. Lines (9) and (10) remove all the new tuples, i.e., the updated versions of the tuples, while lines (11) and (12) restore the tuples as they were before the update. □

**Example 7.15:** An important use of **BEFORE** triggers is to fix up the inserted tuples in some way before they are inserted. Suppose that we want to insert movie tuples into

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

but sometimes, we will not know the year of the movie. Since **year** is part of the primary key, we cannot have **NULL** for this attribute. However, we could make sure that **year** is not **NULL** with a trigger and replace **NULL** by some suitable value, perhaps one that we compute in a complex way. In Fig. 7.7 is a trigger that takes the simple expedient of replacing **NULL** by 1915 (something that could be handled by a default value, but which will serve as an example).

Line (2) says that the condition and action execute before the insertion event. In the referencing-clause of lines (3) through (5), we define names for

```

1) CREATE TRIGGER FixYearTrigger
2) BEFORE INSERT ON Movies
3) REFERENCING
4)     NEW ROW AS NewRow
5)     NEW TABLE AS NewStuff
6) FOR EACH ROW
7) WHEN NewRow.year IS NULL
8) UPDATE NewStuff SET year = 1915;

```

Figure 7.7: Fixing NULL's in inserted tuples

both the new row being inserted and a table consisting of only that row. Even though the trigger executes once for each inserted tuple [because line (6) declares this trigger to be row-level], the condition of line (7) needs to be able to refer to an attribute of the inserted row, while the action of line (8) needs to refer to a table in order to describe an update. □

### 7.5.3 Exercises for Section 7.5

**Exercise 7.5.1:** Write the triggers analogous to Fig. 7.6 for the insertion and deletion events on MovieExec.

**Exercise 7.5.2:** Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the “PC” example of Exercise 2.4.1:

```

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

```

- a) When updating the price of a PC, check that there is no lower priced PC with the same speed.
- b) When inserting a new printer, check that the model number exists in Product.
- c) When making any modification to the Laptop relation, check that the average price of laptops for each manufacturer is at least \$1500.
- d) When updating the RAM or hard disk of any PC, check that the updated PC has at least 100 times as much hard disk as RAM.
- e) When inserting a new PC, laptop, or printer, make sure that the model number did not previously appear in any of PC, Laptop, or Printer.

**Exercise 7.5.3:** Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the battleships example of Exercise 2.4.3.

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

- a) When a new class is inserted into **Classes**, also insert a ship with the name of that class and a NULL launch date.
  - b) When a new class is inserted with a displacement greater than 35,000 tons, allow the insertion, but change the displacement to 35,000.
  - ! c) If a tuple is inserted into **Outcomes**, check that the ship and battle are listed in **Ships** and **Battles**, respectively, and if not, insert tuples into one or both of these relations, with NULL components where necessary.
  - ! d) When there is an insertion into **Ships** or an update of the **class** attribute of **Ships**, check that no country has more than 20 ships.
  - ! e) Check, under all circumstances that could cause a violation, that no ship fought in a battle that was at a later date than another battle in which that ship was sunk.
- ! **Exercise 7.5.4:** Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The problems are based on our running movie example:

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

You may assume that the desired condition holds before any change to the database is attempted. Also, prefer to modify the database, even if it means inserting tuples with NULL or default values, rather than rejecting the attempted modification.

- a) Assure that at all times, any star appearing in **StarsIn** also appears in **MovieStar**.
- b) Assure that at all times every movie executive appears as either a studio president, a producer of a movie, or both.
- c) Assure that every movie has at least one male and one female star.

- d) Assure that the number of movies made by any studio in any year is no more than 100.
- e) Assure that the average length of all movies made in any year is no more than 120.

## 7.6 Summary of Chapter 7

- ◆ *Referential-Integrity Constraints*: We can declare that a value appearing in some attribute or set of attributes must also appear in the corresponding attribute(s) of some tuple of the same or another relation. To do so, we use a REFERENCES or FOREIGN KEY declaration in the relation schema.
- ◆ *Attribute-Based Check Constraints*: We can place a constraint on the value of an attribute by adding the keyword CHECK and the condition to be checked after the declaration of that attribute in its relation schema.
- ◆ *Tuple-Based Check Constraints*: We can place a constraint on the tuples of a relation by adding the keyword CHECK and the condition to be checked to the declaration of the relation itself.
- ◆ *Modifying Constraints*: A tuple-based check can be added or deleted with an ALTER statement for the appropriate table.
- ◆ *Assertions*: We can declare an assertion as an element of a database schema. The declaration gives a condition to be checked. This condition may involve one or more relations of the database schema, and may involve the relation as a whole, e.g., with aggregation, as well as conditions about individual tuples.
- ◆ *Invoking the Checks*: Assertions are checked whenever there is a change to one of the relations involved. Attribute- and tuple-based checks are only checked when the attribute or relation to which they apply changes by insertion or update. Thus, the latter constraints can be violated if they have subqueries.
- ◆ *Triggers*: The SQL standard includes triggers that specify certain events (e.g., insertion, deletion, or update to a particular relation) that awaken them. Once awakened, a condition can be checked, and if true, a specified sequence of actions (SQL statements such as queries and database modifications) will be executed.

## 7.7 References for Chapter 7

References [5] and [4] survey all aspects of active elements in database systems. [1] discusses recent thinking regarding active elements in SQL-99 and future

standards. References [2] and [3] discuss HiPAC, an early prototype system that offered active database elements.

1. R. J. Cochrane, H. Pirahesh, and N. Mattos, “Integrating triggers and declarative constraints in SQL database systems,” *Intl. Conf. on Very Large Database Systems*, pp. 567–579, 1996.
2. U. Dayal et al., “The HiPAC project: combining active databases and timing constraints,” *SIGMOD Record* 17:1, pp. 51–70, 1988.
3. D. R. McCarthy and U. Dayal, “The architecture of an active database management system,” *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 215–224, 1989.
4. N. W. Paton and O. Diaz, “Active database systems,” *Computing Surveys* 31:1 (March, 1999), pp. 63–103.
5. J. Widom and S. Ceri, *Active Database Systems*, Morgan-Kaufmann, San Francisco, 1996.

# Chapter 8

## Views and Indexes

We begin this chapter by introducing virtual views, which are relations that are defined by a query over other relations. Virtual views are not stored in the database, but can be queried as if they existed. The query processor will replace the view by its definition in order to execute the query.

Views can also be materialized, in the sense that they are constructed periodically from the database and stored there. The existence of these materialized views can speed up the execution of queries. A very important specialized type of “materialized view” is the index, a stored data structure whose sole purpose is to speed up the access to specified tuples of one of the stored relations. We introduce indexes here and consider the principal issues in selecting the right indexes for a stored table.

### 8.1 Virtual Views

Relations that are defined with a `CREATE TABLE` statement actually exist in the database. That is, a SQL system stores tables in some physical organization. They are persistent, in the sense that they can be expected to exist indefinitely and not to change unless they are explicitly told to change by a SQL modification statement.

There is another class of SQL relations, called (*virtual*) *views*, that do not exist physically. Rather, they are defined by an expression much like a query. Views, in turn, can be queried as if they existed physically, and in some cases, we can even modify views.

#### 8.1.1 Declaring Views

The simplest form of view definition is:

```
CREATE VIEW <view-name> AS <view-definition>;
```

The view definition is a SQL query.

## Relations, Tables, and Views

SQL programmers tend to use the term “table” instead of “relation.” The reason is that it is important to make a distinction between stored relations, which are “tables,” and virtual relations, which are “views.” Now that we know the distinction between a table and a view, we shall use “relation” only where either a table or view could be used. When we want to emphasize that a relation is stored, rather than a view, we shall sometimes use the term “base relation” or “base table.”

There is also a third kind of relation, one that is neither a view nor stored permanently. These relations are temporary results, as might be constructed for some subquery. Temporaries will also be referred to as “relations” subsequently.

**Example 8.1:** Suppose we want to have a view that is a part of the

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

relation, specifically, the titles and years of the movies made by Paramount Studios. We can define this view by

```

1) CREATE VIEW ParamountMovies AS
2)     SELECT title, year
3)     FROM Movies
4)     WHERE studioName = 'Paramount';

```

First, the name of the view is `ParamountMovies`, as we see from line (1). The attributes of the view are those listed in line (2), namely `title` and `year`. The definition of the view is the query of lines (2) through (4). □

**Example 8.2:** Let us consider a more complicated query used to define a view. Our goal is a relation `MovieProd` with movie titles and the names of their producers. The query defining the view involves two relations:

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

The following view definition

```

CREATE VIEW MovieProd AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;

```

joins the two relations and requires that the certificate numbers match. It then extracts the movie title and producer name from pairs of tuples that agree on the certificates. □

### 8.1.2 Querying Views

A view may be queried exactly as if it were a stored table. We mention its name in a **FROM** clause and rely on the DBMS to produce the needed tuples by operating on the relations used to define the virtual view.

**Example 8.3:** We may query the view **ParamountMovies** just as if it were a stored table, for instance:

```
SELECT title
  FROM ParamountMovies
 WHERE year = 1979;
```

finds the movies made by Paramount in 1979. □

**Example 8.4:** It is also possible to write queries involving both views and base tables. An example is:

```
SELECT DISTINCT starName
  FROM ParamountMovies, StarsIn
 WHERE title = movieTitle AND year = movieYear;
```

This query asks for the name of all stars of movies made by Paramount. □

The simplest way to interpret what a query involving virtual views means is to replace each view in a **FROM** clause by a subquery that is identical to the view definition. That subquery is followed by a tuple variable, so we can refer to its tuples. For instance, the query of Example 8.4 can be thought of as the query of Fig. 8.1.

```
SELECT DISTINCT starName
  FROM (SELECT title, year
        FROM Movies
       WHERE studioName = 'Paramount'
     ) Pm, StarsIn
 WHERE Pm.title = movieTitle AND Pm.year = movieYear;
```

Figure 8.1: Interpreting the use of a virtual view as a subquery

### 8.1.3 Renaming Attributes

Sometimes, we might prefer to give a view's attributes names of our own choosing, rather than use the names that come out of the query defining the view. We may specify the attributes of the view by listing them, surrounded by parentheses, after the name of the view in the **CREATE VIEW** statement. For instance, we could rewrite the view definition of Example 8.2 as:

```
CREATE VIEW MovieProd(movieTitle, prodName) AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```

The view is the same, but its columns are headed by attributes `movieTitle` and `prodName` instead of `title` and `name`.

### 8.1.4 Exercises for Section 8.1

**Exercise 8.1.1:** From the following base tables of our running example

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

Construct the following views:

- A view `RichExec` giving the name, address, certificate number and net worth of all executives with a net worth of at least \$10,000,000.
- A view `StudioPres` giving the name, address, and certificate number of all executives who are studio presidents.
- A view `ExecutiveStar` giving the name, address, gender, birth date, certificate number, and net worth of all individuals who are both executives and stars.

**Exercise 8.1.2:** Write each of the queries below, using one or more of the views from Exercise 8.1.1 and no base tables.

- Find the names of females who are both stars and executives.
- Find the names of those executives who are both studio presidents and worth at least \$10,000,000.
- Find the names of studio presidents who are also stars and are worth at least \$50,000,000.

## 8.2 Modifying Views

In limited circumstances it is possible to execute an insertion, deletion, or update to a view. At first, this idea makes no sense at all, since the view does not exist the way a base table (stored relation) does. What could it mean, say, to insert a new tuple into a view? Where would the tuple go, and how would the database system remember that it was supposed to be in the view?

For many views, the answer is simply “you can’t do that.” However, for sufficiently simple views, called *updatable views*, it is possible to translate the

modification of the view into an equivalent modification on a base table, and the modification can be done to the base table instead. In addition, “instead-of” triggers can be used to turn a view modification into modifications of base tables. In that way, the programmer can force whatever interpretation of a view modification is desired.

### 8.2.1 View Removal

An extreme modification of a view is to delete it altogether. This modification may be done whether or not the view is updatable. A typical **DROP** statement is

```
DROP VIEW ParamountMovies;
```

Note that this statement deletes the definition of the view, so we may no longer make queries or issue modification commands involving this view. However dropping the view does not affect any tuples of the underlying relation **Movies**. In contrast,

```
DROP TABLE Movies
```

would not only make the **Movies** table go away. It would also make the view **ParamountMovies** unusable, since a query that used it would indirectly refer to the nonexistent relation **Movies**.

### 8.2.2 Updatable Views

SQL provides a formal definition of when modifications to a view are permitted. The SQL rules are complex, but roughly, they permit modifications on views that are defined by selecting (using **SELECT**, not **SELECT DISTINCT**) some attributes from one relation *R* (which may itself be an updatable view). Two important technical points:

- The **WHERE** clause must not involve *R* in a subquery.
- The **FROM** clause can only consist of one occurrence of *R* and no other relation.
- The list in the **SELECT** clause must include enough attributes that for every tuple inserted into the view, we can fill the other attributes out with **NULL** values or the proper default. For example, it is not permitted to project out an attribute that is declared **NOT NULL** and has no default.

An insertion on the view can be applied directly to the underlying relation *R*. The only nuance is that we need to specify that the attributes in the **SELECT** clause of the view are the only ones for which values are supplied.

**Example 8.5:** Suppose we insert into view **ParamountMovies** of Example 8.1 a tuple like:

```
INSERT INTO ParamountMovies
VALUES('Star Trek', 1979);
```

View **ParamountMovies** meets the SQL updatability conditions, since the view asks only for some components of some tuples of one base table:

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

The insertion on **ParamountMovies** is executed as if it were the same insertion on **Movies**:

```
INSERT INTO Movies(title, year)
VALUES('Star Trek', 1979);
```

Notice that the attributes **title** and **year** had to be specified in this insertion, since we cannot provide values for other attributes of **Movies**.

The tuple inserted into **Movies** has values 'Star Trek' for **title**, 1979 for **year**, and NULL for the other four attributes. Curiously, the inserted tuple, since it has NULL as the value of attribute **studioName**, will not meet the selection condition for the view **ParamountMovies**, and thus, the inserted tuple has no effect on the view. For instance, the query of Example 8.3 would not retrieve the tuple ('Star Trek', 1979).

To fix this apparent anomaly, we could add **studioName** to the **SELECT** clause of the view, as:

```
CREATE VIEW ParamountMovies AS
  SELECT studioName, title, year
    FROM Movies
   WHERE studioName = 'Paramount';
```

Then, we could insert the *Star-Trek* tuple into the view by:

```
INSERT INTO ParamountMovies
VALUES('Paramount', 'Star Trek', 1979);
```

This insertion has the same effect on **Movies** as:

```
INSERT INTO Movies(studioName, title, year)
VALUES('Paramount', 'Star Trek', 1979);
```

Notice that the resulting tuple, although it has NULL in the attributes not mentioned, does yield the appropriate tuple for the view **ParamountMovies**. □

We may also delete from an updatable view. The deletion, like the insertion, is passed through to the underlying relation  $R$ . However, to make sure that only tuples that can be seen in the view are deleted, we add (using AND) the condition of the WHERE clause in the view to the WHERE clause of the deletion.

**Example 8.6:** Suppose we wish to delete from the updatable Paramount-Movies view all movies with “Trek” in their titles. We may issue the deletion statement

```
DELETE FROM ParamountMovies
WHERE title LIKE '%Trek%';
```

This deletion is translated into an equivalent deletion on the Movies base table; the only difference is that the condition defining the view ParamountMovies is added to the conditions of the WHERE clause.

```
DELETE FROM Movies
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

is the resulting delete statement.  $\square$

Similarly, an update on an updatable view is passed through to the underlying relation. The view update thus has the effect of updating all tuples of the underlying relation that give rise in the view to updated view tuples.

**Example 8.7:** The view update

```
UPDATE ParamountMovies
SET year = 1979
WHERE title = 'Star Trek the Movie';
```

is equivalent to the base-table update

```
UPDATE Movies
SET year = 1979
WHERE title = 'Star Trek the Movie' AND
      studioName = 'Paramount';
```

$\square$

### 8.2.3 Instead-Of Triggers on Views

When a trigger is defined on a view, we can use INSTEAD OF in place of BEFORE or AFTER. If we do so, then when an event awakens the trigger, the action of the trigger is done instead of the event itself. That is, an instead-of trigger intercepts attempts to modify the view and in its place performs whatever action the database designer deems appropriate. The following is a typical example.

## Why Some Views Are Not Updatable

Consider the view `MovieProd` of Example 8.2, which relates movie titles and producers' names. This view is not updatable according to the SQL definition, because there are two relations in the `FROM` clause: `Movies` and `MovieExec`. Suppose we tried to insert a tuple like

```
('Greatest Show on Earth', 'Cecil B. DeMille')
```

We would have to insert tuples into both `Movies` and `MovieExec`. We could use the default value for attributes like `length` or `address`, but what could be done for the two equated attributes `producerC#` and `cert#` that both represent the unknown certificate number of DeMille? We could use `NULL` for both of these. However, when joining relations with `NULL`'s, SQL does not recognize two `NULL` values as equal (see Section 6.1.6). Thus, `'Greatest Show on Earth'` would not be connected with `'Cecil B. DeMille'` in the `MovieProd` view, and our insertion would not have been done correctly.

**Example 8.8:** Let us recall the definition of the view of all movies owned by Paramount:

```
CREATE VIEW ParamountMovies AS
  SELECT title, year
    FROM Movies
   WHERE studioName = 'Paramount';
```

from Example 8.1. As we discussed in Example 8.5, this view is updatable, but it has the unexpected flaw that when you insert a tuple into `ParamountMovies`, the system cannot deduce that the `studioName` attribute is surely Paramount, so `studioName` is `NULL` in the inserted `Movies` tuple.

A better result can be obtained if we create an instead-of trigger on this view, as shown in Fig. 8.2. Much of the trigger is unsurprising. We see the keyword `INSTEAD OF` on line (2), establishing that an attempt to insert into `ParamountMovies` will never take place.

Rather, lines (5) and (6) is the action that replaces the attempted insertion. There is an insertion into `Movies`, and it specifies the three attributes that we know about. Attributes `title` and `year` come from the tuple we tried to insert into the view; we refer to these values by the tuple variable `NewRow` that was declared in line (3) to represent the tuple we are trying to insert. The value of attribute `studioName` is the constant `'Paramount'`. This value is not part of the inserted view tuple. Rather, we assume it is the correct studio for the inserted movie, because the insertion came through the view `ParamountMovies`.

□

```

1) CREATE TRIGGER ParamountInsert
2) INSTEAD OF INSERT ON ParamountMovies
3) REFERENCING NEW ROW AS NewRow
4) FOR EACH ROW
5) INSERT INTO Movies(title, year, studioName)
6) VALUES(NewRow.title, NewRow.year, 'Paramount');

```

Figure 8.2: Trigger to replace an insertion on a view by an insertion on the underlying base table

### 8.2.4 Exercises for Section 8.2

**Exercise 8.2.1:** Which of the views of Exercise 8.1.1 are updatable?

**Exercise 8.2.2:** Suppose we create the view:

```

CREATE VIEW DisneyComedies AS
    SELECT title, year, length FROM Movies
    WHERE studioName = 'Disney' AND genre = 'comedy';

```

- a) Is this view updatable?
- b) Write an instead-of trigger to handle an insertion into this view.
- c) Write an instead-of trigger to handle an update of the length for a movie (given by title and year) in this view.

**Exercise 8.2.3:** Using the base tables

```

Product(maker, model, type)
PC(model, speed, ram, hd, price)

```

suppose we create the view:

```

CREATE VIEW NewPC AS
SELECT maker, model, speed, ram, hd, price
FROM Product, PC
WHERE Product.model = PC.model AND type = 'pc';

```

Notice that we have made a check for consistency: that the model number not only appears in the PC relation, but the `type` attribute of `Product` indicates that the product is a PC.

- a) Is this view updatable?
- b) Write an instead-of trigger to handle an insertion into this view.
- c) Write an instead-of trigger to handle an update of the price.
- d) Write an instead-of trigger to handle a deletion of a specified tuple from this view.

## 8.3 Indexes in SQL

An *index* on an attribute  $A$  of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute  $A$ . We could think of the index as a binary search tree of (key, value) pairs, in which a key  $a$  (one of the values that attribute  $A$  may have) is associated with a “value” that is the set of locations of the tuples that have  $a$  in the component for attribute  $A$ . Such an index may help with queries in which the attribute  $A$  is compared with a constant, for instance  $A = 3$ , or even  $A \leq 3$ . Note that the key for the index can be any attribute or set of attributes, and need not be the key for the relation on which the index is built. We shall refer to the attributes of the index as the *index key* when a distinction needs to be made.

The technology of implementing indexes on large relations is of central importance in the implementation of DBMS’s. The most important data structure used by a typical DBMS is the “B-tree,” which is a generalization of a balanced binary tree. We shall take up B-trees when we talk about DBMS implementation, but for the moment, thinking of indexes as binary search trees will suffice.

### 8.3.1 Motivation for Indexes

When relations are very large, it becomes expensive to scan all the tuples of a relation to find those (perhaps very few) tuples that match a given condition. For example, consider the first query we examined:

```
SELECT *
  FROM Movies
 WHERE studioName = 'Disney' AND year = 1990;
```

from Example 6.1. There might be 10,000 *Movies* tuples, of which only 200 were made in 1990.

The naive way to implement this query is to get all 10,000 tuples and test the condition of the *WHERE* clause on each. It would be much more efficient if we had some way of getting only the 200 tuples from the year 1990 and testing each of them to see if the studio was Disney. It would be even more efficient if we could obtain directly only the 10 or so tuples that satisfied both the conditions of the *WHERE* clause — that the studio is Disney and the year is 1990; see the discussion of “multiatribute indexes,” in Section 8.3.2.

Indexes may also be useful in queries that involve a join. The following example illustrates the point.

**Example 8.9:** Recall the query

```
SELECT name
  FROM Movies, MovieExec
 WHERE title = 'Star Wars' AND producerC# = cert#;
```

from Example 6.12 that asks for the name of the producer of *Star Wars*. If there is an index on **title** of **Movies**, then we can use this index to get the tuple for *Star Wars*. From this tuple, we can extract the **producerC#** to get the certificate of the producer.

Now, suppose that there is also an index on **cert#** of **MovieExec**. Then we can use the **producerC#** with this index to find the tuple of **MovieExec** for the producer of *Star Wars*. From this tuple, we can extract the producer's name. Notice that with these two indexes, we look at only the two tuples, one from each relation, that are needed to answer the query. Without indexes, we have to look at every tuple of the two relations. □

### 8.3.2 Declaring Indexes

Although the creation of indexes is not part of any SQL standard up to and including SQL-99, most commercial systems have a way for the database designer to say that the system should create an index on a certain attribute for a certain relation. The following syntax is typical. Suppose we want to have an index on attribute **year** for the relation **Movies**. Then we say:

```
CREATE INDEX YearIndex ON Movies(year);
```

The result will be that an index whose name is **YearIndex** will be created on attribute **year** of the relation **Movies**. Henceforth, SQL queries that specify a year may be executed by the SQL query processor in such a way that only those tuples of **Movies** with the specified year are ever examined; there is a resulting decrease in the time needed to answer the query.

Often, a DBMS allows us to build a single index on multiple attributes. This type of index takes values for several attributes and efficiently finds the tuples with the given values for these attributes.

**Example 8.10 :** Since **title** and **year** form a key for **Movies**, we might expect it to be common that values for both these attributes will be specified, or neither will. The following is a typical declaration of an index on these two attributes:

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

Since **(title, year)** is a key, it follows that when we are given a title and year, we know the index will find only one tuple, and that will be the desired tuple. In contrast, if the query specifies both the title and year, but only **YearIndex** is available, then the best the system can do is retrieve all the movies of that year and check through them for the given title.

If, as is often the case, the key for the multiatribute index is really the concatenation of the attributes in some order, then we can even use this index to find all the tuples with a given value in the first of the attributes. Thus, part of the design of a multiatribute index is the choice of the order in which the attributes are listed. For instance, if we were more likely to specify a title

than a year for a movie, then we would prefer to order the attributes as above; if a year were more likely to be specified, then we would ask for an index on **(year, title)**. □

If we wish to delete the index, we simply use its name in a statement like:

```
DROP INDEX YearIndex;
```

### 8.3.3 Exercises for Section 8.3

**Exercise 8.3.1:** For our running movies example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Declare indexes on the following attributes or combination of attributes:

- studioName.**
- address of MovieExec.**
- genre and length.**

## 8.4 Selection of Indexes

Choosing which indexes to create requires the database designer to analyze a trade-off. In practice, this choice is one of the principal factors that influence whether a database design gives acceptable performance. Two important factors to consider are:

- The existence of an index on an attribute may speed up greatly the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well.
- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

### 8.4.1 A Simple Cost Model

To understand how to choose indexes for a database, we first need to know where the time is spent answering a query. The details of how relations are stored will be taken up when we consider DBMS implementation. But for the moment, let us state that the tuples of a relation are normally distributed

among many pages of a disk.<sup>1</sup> One page, which is typically several thousand bytes at least, will hold many tuples.

To examine even one tuple requires that the whole page be brought into main memory. On the other hand, it costs little more time to examine all the tuples on a page than to examine only one. There is a great time saving if the page you want is already in main memory, but for simplicity we shall assume that never to be the case, and every page we need must be retrieved from the disk.

### 8.4.2 Some Useful Indexes

Often, the most useful index we can put on a relation is an index on its key. There are two reasons:

1. Queries in which a value for the key is specified are common. Thus, an index on the key will get used frequently.
2. Since there is at most one tuple with a given key value, the index returns either nothing or one location for a tuple. Thus, at most one page must be retrieved to get that tuple into main memory (although there may be other pages that need to be retrieved to use the index itself).

The following example shows the power of key indexes, even in a query that involves a join.

**Example 8.11:** Recall Figure 6.3, where we suggested an exhaustive pairing of tuples of `Movies` and `MovieExec` to compute a join. Implementing the join this way requires us to read each of the pages holding tuples of `Movies` and each of the pages holding tuples of `MovieExec` at least once. In fact, since these pages may be too numerous to fit in main memory at the same time, we may have to read each page from disk many times. With the right indexes, the whole query might be done with as few as two page reads.

An index on the key `title` and `year` for `Movies` would help us find the one `Movies` tuple for *Star Wars* quickly. Only one page — the page containing that tuple — would be read from disk. Then, after finding the producer-certificate number in that tuple, an index on the key `cert#` for `MovieExec` would help us quickly find the one tuple for the producer in the `MovieExec` relation. Again, only one page with `MovieExec` tuples would be read from disk, although we might need to read a small number of other pages to use the `cert#` index. □

When the index is not on a key, it may or may not be able to improve the time spent retrieving from disk the tuples needed to answer a query. There are two situations in which an index can be effective, even if it is not on a key.

---

<sup>1</sup>Pages are usually referred to as “blocks” in discussion of databases, but if you are familiar with a paged-memory system from operating systems you should think of the disk as divided into pages.

1. If the attribute is almost a key; that is, relatively few tuples have a given value for that attribute. Even if each of the tuples with a given value is on a different page, we shall not have to retrieve many pages from disk.
2. If the tuples are “clustered” on that attribute. We *cluster* a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible. Then, even if there are many tuples, we shall not have to retrieve nearly as many pages as there are tuples.

**Example 8.12:** As an example of an index of the first kind, suppose **Movies** had an index on **title** rather than **title** and **year**. Since **title** by itself is not a key for the relation, there would be titles such as *King Kong*, where several tuples matched the index key **title**. If we compared use of the index on **title** with what happens in Example 8.11, we would find that a search for movies with title *King Kong* would produce three tuples (because there are three movies with that title, from years 1933, 1976, and 2005). It is possible that these tuples are on three different pages, so all three pages would be brought into main memory, roughly tripling the amount of time this step takes. However, since the relation **Movies** probably is spread over many more than three pages, there is still a considerable time saving in using the index.

At the next step, we need to get the three **producerC#** values from these three tuples, and find in the relation **MovieExec** the producers of these three movies. We can use the index on **cert#** to find the three relevant tuples of **MovieExec**. Possibly they are on three different pages, but we still spend less time than we would if we had to bring the entire **MovieExec** relation into main memory. □

**Example 8.13:** Now, suppose the only index we have on **Movies** is one on **year**, and we want to answer the query:

```
SELECT *
FROM Movies
WHERE year = 1990;
```

First, suppose the tuples of **Movies** are not clustered by **year**; say they are stored alphabetically by **title**. Then this query gains little from the index on **year**. If there are, say, 100 movies per page, there is a good chance that any given page has at least one movie made in 1990. Thus, a large fraction of the pages used to hold the relation **Movies** will have to be brought to main memory.

However, suppose the tuples of **Movies** are clustered on **year**. Then we could use the index on **year** to find only the small number of pages that contained tuples with **year** = 1990. In this case, the **year** index will be of great help. In comparison, an index on the combination of **title** and **year** would be of little help, no matter what attribute or attributes we used to cluster **Movies**. □

### 8.4.3 Calculating the Best Indexes to Create

It might seem that the more indexes we create, the more likely it is that an index useful for a given query will be available. However, if modifications are the most frequent action, then we should be very conservative about creating indexes. Each modification on a relation  $R$  forces us to change any index on one or more of the modified attributes of  $R$ . Thus, we must read and write not only the pages of  $R$  that are modified, but also read and write certain pages that hold the index. But even when modifications are the dominant form of database action, it may be an efficiency gain to create an index on a frequently used attribute. In fact, since some modification commands involve querying the database (e.g., an `INSERT` with a select-from-where subquery or a `DELETE` with a condition) one must be very careful how one estimates the relative frequency of modifications and queries.

Remember that the typical relation is stored over many disk blocks (pages), and the principal cost of a query or modification is often the number of pages that need to be brought to main memory. Thus, indexes that let us find a tuple without examining the entire relation can save a lot of time. However, the indexes themselves have to be stored, at least partially, on disk, so accessing and modifying the indexes themselves cost disk accesses. In fact, modification, since it requires one disk access to read a page and another disk access to write the changed page, is about twice as expensive as accessing the index or the data in a query.

To calculate the new value of an index, we need to make assumptions about which queries and modifications are most likely to be performed on the database. Sometimes, we have a history of queries that we can use to get good information, on the assumption that the future will be like the past. In other cases, we may know that the database supports a particular application or applications, and we can see in the code for those applications all the SQL queries and modifications that they will ever do. In either situation, we are able to list what we expect are the most common query and modification forms. These forms can have variables in place of constants, but should otherwise look like real SQL statements. Here is a simple example of the process, and of the calculations that we need to make.

**Example 8.14:** Let us consider the relation

```
StarsIn(movieTitle, movieYear, starName)
```

Suppose that there are three database operations that we sometimes perform on this relation:

$Q_1$ : We look for the title and year of movies in which a given star appeared. That is, we execute a query of the form:

```
SELECT movieTitle, movieYear  
FROM StarsIn  
WHERE starName = s;
```

for some constant  $s$ .

- $Q_2$ : We look for the stars that appeared in a given movie. That is, we execute a query of the form:

```
SELECT starName
  FROM StarsIn
 WHERE movieTitle = t AND movieYear = y;
```

for constants  $t$  and  $y$ .

- $I$ : We insert a new tuple into StarsIn. That is, we execute an insertion of the form:

```
INSERT INTO StarsIn VALUES(t, y, s);
```

for constants  $t$ ,  $y$ , and  $s$ .

Let us make the following assumptions about the data:

1. StarsIn occupies 10 pages, so if we need to examine the entire relation the cost is 10.
2. On the average, a star has appeared in 3 movies and a movie has 3 stars.
3. Since the tuples for a given star or a given movie are likely to be spread over the 10 pages of StarsIn, even if we have an index on **starName** or on the combination of **movieTitle** and **movieYear**, it will take 3 disk accesses to find the (average of) 3 tuples for a star or movie. If we have no index on the star or movie, respectively, then 10 disk accesses are required.
4. One disk access is needed to read a page of the index every time we use that index to locate tuples with a given value for the indexed attribute(s). If an index page must be modified (in the case of an insertion), then another disk access is needed to write back the modified page.
5. Likewise, in the case of an insertion, one disk access is needed to read a page on which the new tuple will be placed, and another disk access is needed to write back this page. We assume that, even without an index, we can find some page on which an additional tuple will fit, without scanning the entire relation.

Figure 8.3 gives the costs of each of the three operations;  $Q_1$  (query given a star),  $Q_2$  (query given a movie), and  $I$  (insertion). If there is no index, then we must scan the entire relation for  $Q_1$  or  $Q_2$  (cost 10),<sup>2</sup> while an insertion requires

---

<sup>2</sup>There is a subtle point that we shall ignore here. In many situations, it is possible to store a relation on disk using consecutive pages or tracks. In that case, the cost of retrieving the entire relation may be significantly less than retrieving the same number of pages chosen randomly.

Action	No Index	Star Index	Movie Index	Both Indexes
$Q_1$	10	4	10	4
$Q_2$	10	10	4	4
$I$	2	4	4	6
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

Figure 8.3: Costs associated with the three actions, as a function of which indexes are selected

merely that we access a page with free space and rewrite it with the new tuple (cost of 2, since we assume that page can be found without an index). These observations explain the column labeled “No Index.”

If there is an index on stars only, then  $Q_2$  still requires a scan of the entire relation (cost 10). However,  $Q_1$  can be answered by accessing one index page to find the three tuples for a given star and then making three more accesses to find those tuples. Insertion  $I$  requires that we read and write both a page for the index and a page for the data, for a total of 4 disk accesses.

The case where there is an index on movies only is symmetric to the case for stars only. Finally, if there are indexes on both stars and movies, then it takes 4 disk accesses to answer either  $Q_1$  or  $Q_2$ . However, insertion  $I$  requires that we read and write two index pages as well as a data page, for a total of 6 disk accesses. That observation explains the last column in Fig. 8.3.

The final row in Fig. 8.3 gives the average cost of an action, on the assumption that the fraction of the time we do  $Q_1$  is  $p_1$  and the fraction of the time we do  $Q_2$  is  $p_2$ ; therefore, the fraction of the time we do  $I$  is  $1 - p_1 - p_2$ .

Depending on  $p_1$  and  $p_2$ , any of the four choices of index/no index can yield the best average cost for the three actions. For example, if  $p_1 = p_2 = 0.1$ , then the expression  $2 + 8p_1 + 8p_2$  is the smallest, so we would prefer not to create any indexes. That is, if we are doing mostly insertion, and very few queries, then we don’t want an index. On the other hand, if  $p_1 = p_2 = 0.4$ , then the formula  $6 - 2p_1 - 2p_2$  turns out to be the smallest, so we would prefer indexes on both `starName` and on the (`movieTitle`, `movieYear`) combination. Intuitively, if we are doing a lot of queries, and the number of queries specifying movies and stars are roughly equally frequent, then both indexes are desired.

If we have  $p_1 = 0.5$  and  $p_2 = 0.1$ , then an index on stars only gives the best average value, because  $4 + 6p_2$  is the formula with the smallest value. Likewise,  $p_1 = 0.1$  and  $p_2 = 0.5$  tells us to create an index on only movies. The intuition is that if only one type of query is frequent, create only the index that helps that type of query.  $\square$

#### 8.4.4 Automatic Selection of Indexes to Create

“Tuning” a database is a process that includes not only index selection, but the choice of many different parameters. We have not yet discussed much about

physical implementation of databases, but some examples of tuning issues are the amount of main memory to allocate to various processes and the rate at which backups and checkpoints are made (to facilitate recovery from a crash). There are a number of tools that have been designed to take the responsibility from the database designer and have the system tune itself, or at least advise the designer on good choices.

We shall mention some of these projects in the bibliographic notes for this chapter. However, here is an outline of how the index-selection portion of tuning advisors work.

1. The first step is to establish the query workload. Since a DBMS normally logs all operations anyway, we may be able to examine the log and find a set of representative queries and database modifications for the database at hand. Or it is possible that we know, from the application programs that use the database, what the typical queries will be.
2. The designer may be offered the opportunity to specify some constraints, e.g., indexes that must, or must not, be chosen.
3. The tuning advisor generates a set of possible *candidate* indexes, and evaluates each one. Typical queries are given to the query optimizer of the DBMS. The query optimizer has the ability to estimate the running times of these queries under the assumption that one particular set of indexes is available.
4. The index set resulting in the lowest cost for the given workload is suggested to the designer, or it is automatically created.

A subtle issue arises when we consider possible indexes in step (3). The existence of previously chosen indexes may influence how much *benefit* (improvement in average execution time of the query mix) another index offers. A “greedy” approach to choosing indexes has proven effective.

- a) Initially, with no indexes selected, evaluate the benefit of each of the candidate indexes. If at least one provides positive benefit (i.e., it reduces the average execution time of queries), then choose that index.
- b) Then, reevaluate the benefit of each of the remaining candidate indexes, assuming that the previously selected index is also available. Again, choose the index that provides the greatest benefit, assuming that benefit is positive.
- c) In general, repeat the evaluation of candidate indexes under the assumption that all previously selected indexes are available. Pick the index with maximum benefit, until no more positive benefits can be obtained.

### 8.4.5 Exercises for Section 8.4

**Exercise 8.4.1:** Suppose that the relation `StarsIn` discussed in Example 8.14 required 100 pages rather than 10, but all other assumptions of that example continued to hold. Give formulas in terms of  $p_1$  and  $p_2$  to measure the cost of queries  $Q_1$  and  $Q_2$  and insertion  $I$ , under the four combinations of index/no index discussed there.

**Exercise 8.4.2:** In this problem, we consider indexes for the relation

`Ships(name, class, launched)`

from our running battleships exercise. Assume:

- i. `name` is the key.
- ii. The relation `Ships` is stored over 50 pages.
- iii. The relation is clustered on `class` so we expect that only one disk access is needed to find the ships of a given class.
- iv. On average, there are 5 ships of a class, and 25 ships launched in any given year.
- v. With probability  $p_1$  the operation on this relation is a query of the form  
`SELECT * FROM Ships WHERE name = n.`
- vi. With probability  $p_2$  the operation on this relation is a query of the form  
`SELECT * FROM Ships WHERE class = c.`
- vii. With probability  $p_3$  the operation on this relation is a query of the form  
`SELECT * FROM Ships WHERE launched = y.`
- viii. With probability  $1 - p_1 - p_2 - p_3$  the operation on this relation is an insertion of a new tuple into `Ships`.

You can also make the assumptions about accessing indexes and finding empty space for insertions that were made in Example 8.14.

Consider the creation of indexes on `name`, `class`, and `launched`. For each combination of indexes, estimate the average cost of an operation. As a function of  $p_1$ ,  $p_2$ , and  $p_3$ , what is the best choice of indexes?

## 8.5 Materialized Views

A view describes how a new relation can be constructed from base tables by executing a query on those tables. Until now, we have thought of views only as logical descriptions of relations. However, if a view is used frequently enough, it may even be efficient to *materialize* it; that is, to maintain its value at all times. As with maintaining indexes, there is a cost involved in maintaining a materialized view, since we must recompute parts of the materialized view each time one of the underlying base tables changes.

### 8.5.1 Maintaining a Materialized View

In principle, the DBMS needs to recompute a materialized view every time one of its base tables changes in any way. For simple views, it is possible to limit the number of times we need to consider changing the materialized view, and it is possible to limit the amount of work we do when we must maintain the view. We shall take up an example of a join view, and see that there are a number of opportunities to simplify our work.

**Example 8.15:** Suppose we frequently want to find the name of the producer of a given movie. We might find it advantageous to materialize a view:

```
CREATE MATERIALIZED VIEW MovieProd AS
  SELECT title, year, name
    FROM Movies, MovieExec
   WHERE producerC# = cert#
```

To start, the DBMS does not have to consider the effect on **MovieProd** of an update on any attribute of **Movies** or **MovieExec** that is not mentioned in the query that defines the materialized view. Surely any modification to a relation that is neither **Movies** nor **MovieExec** can be ignored as well. However, there are a number of other simplifications that enable us to handle other modifications to **Movies** or **MovieExec** more efficiently than a re-execution of the query that defines the materialized view.

1. Suppose we insert a new movie into **Movies**, say **title** = 'Kill Bill', **year** = 2003, and **producerC#** = 23456. Then we only need to look up **cert#** = 23456 in **MovieExec**. Since **cert#** is the key for **MovieExec**, there can be at most one name returned by the query

```
SELECT name FROM MovieExec
  WHERE cert# = 23456;
```

As this query returns **name** = 'Quentin Tarantino', the DBMS can insert the proper tuple into **MovieProd** by:

```
INSERT INTO MovieProd
  VALUES('Kill Bill', 2003, 'Quentin Tarantino');
```

Note that, since **MovieProd** is materialized, it is stored like any base table, and this operation makes sense; it does not have to be reinterpreted by an instead-of trigger or any other mechanism.

2. Suppose we delete a movie from **Movies**, say the movie with **title** = 'Dumb & Dumber' and **year** = 1994. The DBMS has only to delete this one movie from **MovieProd** by:

```
DELETE FROM MovieProd
WHERE title = 'Dumb & Dumber' AND year = 1994;
```

3. Suppose we insert a tuple into `MovieExec`, and that tuple has `cert# = 34567` and `name = 'Max Bialystock'`. Then the DBMS may have to insert into `MovieProd` some movies that were not there because their producer was previously unknown. The operation is:

```
INSERT INTO MovieProd
SELECT title, year, 'Max Bialystock'
FROM Movies
WHERE producerC# = 34567;
```

4. Suppose we delete the tuple with `cert# = 45678` from `MovieExec`. Then the DBMS must delete from `MovieProd` all movies that have `producerC# = 45678`, because there now can be no matching tuple in `MovieExec` for their underlying `Movies` tuple. Thus, the DBMS executes:

```
DELETE FROM MovieProd
WHERE (title, year) IN
(SELECT title, year FROM Movies
 WHERE producerC# = 45678);
```

Notice that it is not sufficient to look up the `name` corresponding to 45678 in `MovieExec` and delete all movies from `MovieProd` that have that producer name. The reason is that, because `name` is not a key for `MovieExec`, there could be two producers with the same name.

We leave as an exercise the consideration of how updates to `Movies` that involve `title` or `year` are handled, and how updates to `MovieExec` involving `cert#` are handled. □

The most important thing to take away from Example 8.15 is that all the changes to the materialized view are *incremental*. That is, we never have to reconstruct the whole view from scratch. Rather, insertions, deletions, and updates to a base table can be implemented in a join view such as `MovieProd` by a small number of queries to the base tables followed by modification statements on the materialized view. Moreover, these modifications do not affect all the tuples of the view, but only those that have at least one attribute with a particular constant.

It is not possible to find rules such as those in Example 8.15 for any materialized view we could construct; some are just too complicated. However, many common types of materialized view *do* allow the view to be maintained incrementally. We shall explore another common type of materialized view — aggregation views — in the exercises.

### 8.5.2 Periodic Maintenance of Materialized Views

There is another setting in which we may use materialized views, yet not have to worry about the cost or complexity of maintaining them up-to-date as the underlying base tables change. We shall encounter the option when we study OLAP in Section 10.6, but for the moment let us remark that it is common for databases to serve two purposes. For example, a department store may use its database to record its current inventory; this data changes with every sale. The same database may be used by analysts to study buyer patterns and to predict when the store is going to need to restock an item.

The analysts' queries may be answered more efficiently if they can query materialized views, especially views that aggregate data (e.g., sum the inventories of different sizes of shirt after grouping by style). But the database is updated with each sale, so modifications are far more frequent than queries. When modifications dominate, it is costly to have materialized views, or even indexes, on the data.

What is usually done is to create materialized views, but not to try to keep them up-to-date as the base tables change. Rather, the materialized views are reconstructed periodically (typically each night), when other activity in the database is low. The materialized views are only used by analysts, and their data might be out of date by as much as 24 hours. However, in normal situations, the rate at which an item is bought by customers changes slowly. Thus, the data will be "good enough" for the analysts to predict items that are selling well and those that are selling poorly. Of course if Brad Pitt is seen wearing a Hawaiian shirt one morning, and every cool guy has to buy one by that evening, the analysts will not notice they are out of Hawaiian shirts until the next morning, but the risk of that sort of occurrence is low.

### 8.5.3 Rewriting Queries to Use Materialized Views

A materialized view can be referred to in the `FROM` clause of a query, just as a virtual view can (Section 8.1.2). However, because a materialized view is stored in the database, it is possible to rewrite a query to use a materialized view, even if that view was not mentioned in the query as written. Such a rewriting may enable the query to execute much faster, because the hard parts of the query, e.g., joining of relations, may have been carried out already when the materialized view was constructed.

However, we must be very careful to check that the query can be rewritten to use a materialized view. A complete set of rules that will let us use materialized views of any kind is beyond the scope of this book. However, we shall offer a relatively simple rule that applies to the view of Example 8.15 and similar views.

Suppose we have a materialized view  $V$  defined by a query of the form:

```
SELECT  $L_V$ 
  FROM  $R_V$ 
 WHERE  $C_V$ 
```

where  $L_V$  is a list of attributes,  $R_V$  is a list of relations, and  $C_V$  is a condition. Similarly, suppose we have a query  $Q$  of the same form:

```
SELECT  $L_Q$ 
  FROM  $R_Q$ 
 WHERE  $C_Q$ 
```

Here are the conditions under which we can replace part of the query  $Q$  by the view  $V$ .

1. The relations in list  $R_V$  all appear in the list  $R_Q$ .
2. The condition  $C_Q$  is equivalent to  $C_V \text{ AND } C$  for some condition  $C$ . As a special case,  $C_Q$  could be equivalent to  $C_V$ , in which case the “AND  $C$ ” is unnecessary.
3. If  $C$  is needed, then the attributes of relations on list  $R_V$  that  $C$  mentions are attributes on the list  $L_V$ .
4. Attributes on the list  $L_Q$  that come from relations on the list  $R_V$  are also on the list  $L_V$ .

If all these conditions are met, then we can rewrite  $Q$  to use  $V$ , as follows:

- a) Replace the list  $R_Q$  by  $V$  and the relations that are on list  $R_Q$  but not on  $R_V$ .
- b) Replace  $C_Q$  by  $C$ . If  $C$  is not needed (i.e.,  $C_V = C_Q$ ), then there is no WHERE clause.

**Example 8.16:** Suppose we have the materialized view `MovieProd` from Example 8.15. This view is defined by the query  $V$ :

```
SELECT title, year, name
  FROM Movies, MovieExec
 WHERE producerC# = cert#
```

Suppose also that we need to answer the query  $Q$  that asks for the names of the stars of movies produced by Max Bialystock. For this query we need the relations:

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

The query  $Q$  can be written:

```
SELECT starName
  FROM StarsIn, Movies, MovieExec
 WHERE movieTitle = title AND movieYear = year AND
       producerC# = cert# AND name = 'Max Bialystock';
```

Let us compare the view definition  $V$  with the query  $Q$ , to see that they meet the conditions listed above.

1. The relations in the `FROM` clause of  $V$  are all in the `FROM` clause of  $Q$ .
2. The condition from  $Q$  can be written as the condition from  $V \text{ AND } C$ , where  $C =$

```
movieTitle = title AND movieYear = year AND
name = 'Max Bialystock'
```

3. The attributes of  $C$  that come from relations of  $V$  (`Movies` and `MovieExec`) are `title`, `year`, and `name`. These attributes all appear in the `SELECT` clause of  $V$ .
4. No attribute from the `SELECT` list of  $Q$  is from a relation that appears in the `FROM` list of  $V$ .

We may thus use  $V$  in  $Q$ , yielding the rewritten query:

```
SELECT starName
FROM StarsIn, MovieProd
WHERE movieTitle = title AND movieYear = year AND
      name = 'Max Bialystock';
```

That is, we replaced `Movies` and `MovieExec` in the `FROM` clause by the materialized view `MovieProd`. We also removed the condition of the view from the `WHERE` clause, leaving only the condition  $C$ . Since the rewritten query involves the join of only two relations, rather than three, we expect the rewritten query to execute in less time than the original.  $\square$

### 8.5.4 Automatic Creation of Materialized Views

The ideas that were discussed in Section 8.4.4 for indexes can apply as well to materialized views. We first need to establish or approximate the query workload. An automated materialized-view-selection advisor needs to generate candidate views. This task can be far more difficult than generating candidate indexes. In the case of indexes, there is only one possible index for each attribute of each relation. We could also consider indexes on small sets of attributes of a relation, but even if we do, generating all the candidate indexes is straightforward. However, with materialized views, any query could in principle define a view, so there is no limit on what views we need to consider.

The process can be limited if we remember that there is no point in creating a materialized view that does not help for at least one query of our expected workload. For example, suppose some or all of the queries in our workload have the form considered in Section 8.5.3. Then we can use the analysis of that section to find the views that can help a given query. We can limit ourselves to candidate materialized views that:

1. Have a list of relations in the **FROM** clause that is a subset of those in the **FROM** clause of at least one query of the workload.
2. Have a **WHERE** clause that is the **AND** of conditions that each appear in at least one query.
3. Have a list of attributes in the **SELECT** clause that is sufficient to be used in at least one query.

To evaluate the benefit of a materialized view, let the query optimizer estimate the running times of the queries, both with and without the materialized view. Of course, the optimizer must be designed to take advantage of materialized views; all modern optimizers know how to exploit indexes, but not all can exploit materialized views. Section 8.5.3 was an example of the reasoning that would be necessary for a query optimizer to perform, if it were to take advantage of such views.

There is another issue that comes up when we consider automatic choice of materialized views, but that did not surface for indexes. An index on a relation is generally smaller than the relation itself, and all indexes on one relation take roughly the same amount of space. However, materialized views can vary radically in size, and some — those involving joins — can be very much larger than the relation or relations on which they are built. Thus, we may need to rethink the definition of the “benefit” of a materialized view. For example, we might want to define the benefit to be the improvement in average running time of the query workload divided by the amount of space the view occupies.

### 8.5.5 Exercises for Section 8.5

**Exercise 8.5.1:** Complete Example 8.15 by considering updates to either of the base tables.

! **Exercise 8.5.2:** Suppose the view **NewPC** of Exercise 8.2.3 were a materialized view. What modifications to the base tables **Product** and **PC** would require a modification of the materialized view? How would you implement those modifications incrementally?

! **Exercise 8.5.3:** This exercise explores materialized views that are based on aggregation of data. Suppose we build a materialized view on the base tables

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

from our running battleships exercise, as follows:

```
CREATE MATERIALIZED VIEW ShipStats AS
    SELECT country, AVG(displacement), COUNT(*)
    FROM Classes, Ships
    WHERE Classes.class = Ships.class
    GROUP BY country;
```

What modifications to the base tables `Classes` and `Ships` would require a modification of the materialized view? How would you implement those modifications incrementally?

! **Exercise 8.5.4:** In Section 8.5.3 we gave conditions under which a materialized view of simple form could be used in the execution of a query of similar form. For the view of Example 8.15, describe all the queries of that form, for which this view could be used.

## 8.6 Summary of Chapter 8

- ◆ *Virtual Views*: A virtual view is a definition of how one relation (the view) may be constructed logically from tables stored in the database or other views. Views may be queried as if they were stored relations. The query processor modifies queries about a view so the query is instead about the base tables that are used to define the view.
- ◆ *Updatable Views*: Some virtual views on a single relation are updatable, meaning that we can insert into, delete from, and update the view as if it were a stored table. These operations are translated into equivalent modifications to the base table over which the view is defined.
- ◆ *Instead-Of Triggers*: SQL allows a special type of trigger to apply to a virtual view. When a modification to the view is called for, the instead-of trigger turns the modification into operations on base tables that are specified in the trigger.
- ◆ *Indexes*: While not part of the SQL standard, commercial SQL systems allow the declaration of indexes on attributes; these indexes speed up certain queries or modifications that involve specification of a value, or range of values, for the indexed attribute(s).
- ◆ *Choosing Indexes*: While indexes speed up queries, they slow down database modifications, since the indexes on the modified relation must also be modified. Thus, the choice of indexes is a complex problem, depending on the actual mix of queries and modifications performed on the database.
- ◆ *Automatic Index Selection*: Some DBMS's offer tools that choose indexes for a database automatically. They examine the typical queries and modifications performed on the database and evaluate the cost trade-offs for different indexes that might be created.
- ◆ *Materialized Views*: Instead of treating a view as a query on base tables, we can use the query as a definition of an additional stored relation, whose value is a function of the values of the base tables.

- ◆ **Maintaining Materialized Views:** As the base tables change, we must make the corresponding changes to any materialized view whose value is affected by the change. For many common kinds of materialized views, it is possible to make the changes to the view incrementally, without recomputing the entire view.
- ◆ **Rewriting Queries to Use Materialized Views:** The conditions under which a query can be rewritten to use a materialized view are complex. However, if the query optimizer can perform such rewritings, then an automatic design tool can consider the improvement in performance that results from creating materialized views and can select views to materialize, automatically.

## 8.7 References for Chapter 8

The technology behind materialized views is surveyed in [2] and [7]. Reference [3] introduces the greedy algorithm for selecting materialized views.

Two projects for automatically tuning databases are AutoAdmin at Microsoft and SMART at IBM. Current information on AutoAdmin can be found on-line at [8]. A description of the technology behind this system is in [1].

A survey of the SMART project is in [4]. The index-selection aspect of the project is described in [6].

Reference [5] surveys index selection, materialized views, automatic tuning, and related subjects covered in this chapter.

1. S. Agrawal, S. Chaudhuri, and V. R. Narasayya, “Automated selection of materialized views and indexes in SQL databases,” *Proc. Intl. Conference on Very Large Databases*, pp. 496–505, 2000.
2. A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, Cambridge MA, 1999.
3. V. Harinarayan, A. Rajaraman, and J. D. Ullman, “Implementing data cubes efficiently,” *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1996), pp. 205–216.
4. S. S. Lightstone, G. Lohman, and S. Zilio, “Toward autonomic computing with DB2 universal database,” *SIGMOD Record* 31:3, pp. 55–61, 2002.
5. S. S. Lightstone, T. Teorey, and T. Nadeau, *Physical Database Design*, Morgan-Kaufmann, San Francisco, 2007.
6. G. Lohman, G. Valentin, D. Zilio, M. Zuliani, and A. Skelley, “DB2 Advisor: an optimizer smart enough to recommend its own indexes,” *Proc. Sixteenth IEEE Conf. on Data Engineering*, pp. 101–110, 2000.
7. D. Lomet and J. Widom (eds.), Special issue on materialized views and data warehouses, *IEEE Data Engineering Bulletin* 18:2 (1995).

8. Microsoft on-line description of the AutoAdmin project.

<http://research.microsoft.com/dmx/autoadmin/>

# Chapter 9

# SQL in a Server Environment

We now turn to the question of how SQL fits into a complete programming environment. The typical server environment is introduced in Section 9.1. Section 9.2 introduces the SQL terminology for client-server computing and connecting to a database.

Then, we turn to how programming is really done, when SQL must be used to access a database as part of a typical application. In Section 9.3 we see how to embed SQL in programs that are written in an ordinary programming language, such as C. A critical issue is how we move data between SQL relations and the variables of the surrounding, or “host,” language. Section 9.4 considers another way to combine SQL with general-purpose programming: persistent stored modules, which are pieces of code stored as part of a database schema and executable on command from the user.

A third programming approach is a “call-level interface,” where we program in some conventional language and use a library of functions to access the database. In Section 9.5 we discuss the SQL-standard library called SQL/CLI, for making calls from C programs. Then, in Section 9.6 we meet Java’s JDBC (database connectivity), which is an alternative call-level interface. Finally, another popular call-level interface, PHP, is covered in Section 9.7.

## 9.1 The Three-Tier Architecture

Databases are used in many different settings, including small, standalone databases. For example, a scientist may run a copy of MySQL or Microsoft Access on a laboratory computer to store experimental data. However, there is a very common architecture for large database installations; this architecture motivates the discussion of the entire chapter. The architecture is called *three-tier* or *three-layer*, because it distinguishes three different, interacting functions:

1. *Web Servers.* These are processes that connect clients to the database system, usually over the Internet or possibly a local connection.
2. *Application Servers.* These processes perform the “business logic,” whatever it is the system is intended to do.
3. *Database Servers.* These processes run the DBMS and perform queries and modifications at the request of the application servers.

The processes may all run on the same processor in a small system, but it is common to dedicate a large number of processors to each of the tiers. Figure 9.1 suggests how a large database installation would be organized.

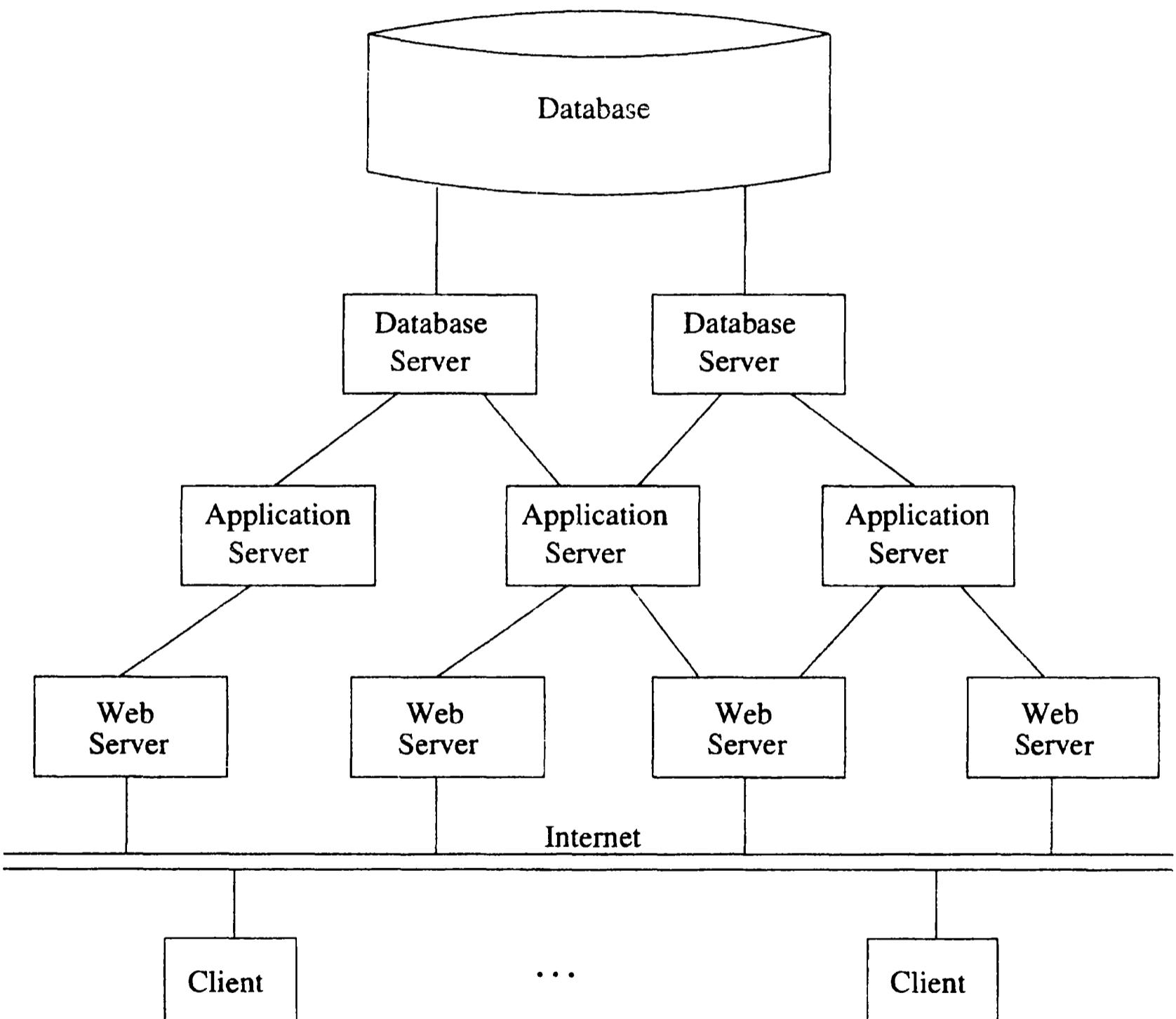


Figure 9.1: The Three-Tier Architecture

### 9.1.1 The Web-Server Tier

The web-server processes manage the interactions with the user. When a user makes contact, perhaps by opening a URL, a web server, typically running

Apache/Tomcat, responds to the request. The user then becomes a *client* of this web-server process. Typically, the client's actions are performed by the web-browser, e.g., managing of the filling of forms, which are then posted to the web server.

As an example, let us consider a site such as Amazon.com. A user (customer) opens a connection to the Amazon database system by entering the URL `www.amazon.com` into their browser. The Amazon web-server presents a “home page” to the user, which includes forms, menus, and buttons enabling the user to express what it is they want to do. For example, the user may set a menu to Books and enter into a form the title of the book they are interested in. The client web-browser transmits this information to the Amazon web-server, and that web-server must negotiate with the next tier — the application tier — to fulfill the client's request.

### 9.1.2 The Application Tier

The job of the application tier is to turn data, from the database, into a response to the request that it receives from the web-server. Each web-server process can invoke one or more application-tier processes to handle the request; these processes can be on one machine or many, and they may be on the same or different machines from the web-server processes.

The actions performed by the application tier are often referred to as the *business logic* of the organization operating the database. That is, one designs the application tier by reasoning out what the response to a request by the potential customer should be, and then implementing that strategy.

In the case of our example of a book at Amazon.com, this response would be the elements of the page that Amazon displays about a book. That data includes the title, author, price, and several other pieces of information about the book. It also includes links to more information, such as reviews, alternative sellers of the book, and similar books.

In a simple system, the application tier may issue database queries directly to the database tier, and assemble the results of those queries, perhaps in an HTML page. In a more complex system, there can be several subtiers to the application tier, and each may have its own processes. A common architecture is to have a subtier that supports “objects.” These objects can contain data such as the title and price of a book in a “book object.” Data for this object is obtained by a database query. The object may also have methods that can be invoked by the application-tier processes, and these methods may in turn cause additional queries to be issued to the database when and if they are invoked.

Another subtier may be present to support *database integration*. That is, there may be several quite independent databases that support operations, and it may not be possible to issue queries involving data from more than one database at a time. The results of queries to different sources may need to be combined at the integration subtier. To make integration more complex, the databases may not be compatible in a number of important ways. We shall

examine the technology of information integration elsewhere. However, for the moment, consider the following hypothetical example.

**Example 9.1:** The Amazon database containing information about a book may have a price in dollars. But the customer is in Europe, and their account information is in another database, located in Europe, with billing information in Euros. The integration subtier needs to know that there is a difference in currencies, when it gets a price from the books database and uses that price to enter data into a bill that is displayed to the customer. □

### 9.1.3 The Database Tier

Like the other tiers, there can be many processes in the database tier, and the processes can be distributed over many machines, or all be together on one. The database tier executes queries that are requested from the application tier, and may also provide some buffering of data. For example, a query that produces many tuples may be fed one-at-a-time to the requesting process of the application tier.

Since creating connections to the database takes significant time, we normally keep a large number of connections open and allow application processes to share these connections. Each application process must return the connection to the state in which it was found, to avoid unexpected interactions between application processes.

The balance of this chapter is about how we implement a database tier. Especially, we need to learn:

1. How do we enable a database to interact with “ordinary” programs that are written in a conventional language such as C or Java?
2. How do we deal with the differences in data-types supported by SQL and conventional languages? In particular, relations are the results of queries, and these are not directly supported by conventional languages.
3. How do we manage connections to a database when these connections are shared between many short-lived processes?

## 9.2 The SQL Environment

In this section we shall take the broadest possible view of a DBMS and the databases and programs it supports. We shall see how databases are defined and organized into clusters, catalogs, and schemas. We shall also see how programs are linked with the data they need to manipulate. Many of the details depend on the particular implementation, so we shall concentrate on the general ideas that are contained in the SQL standard. Sections 9.5, 9.6, and 9.7 illustrate how these high-level concepts appear in a “call-level interface,” which requires the programmer to make explicit connections to databases.

### 9.2.1 Environments

A *SQL environment* is the framework under which data may exist and SQL operations on data may be executed. In practice, we should think of a SQL environment as a DBMS running at some installation. For example, ABC company buys a license for the Megatron 2010 DBMS to run on a collection of ABC's machines. The system running on these machines constitutes a SQL environment.

All the database elements we have discussed — tables, views, triggers, and so on — are defined within a SQL environment. These elements are organized into a hierarchy of structures, each of which plays a distinct role in the organization. The structures defined by the SQL standard are indicated in Fig. 9.2.

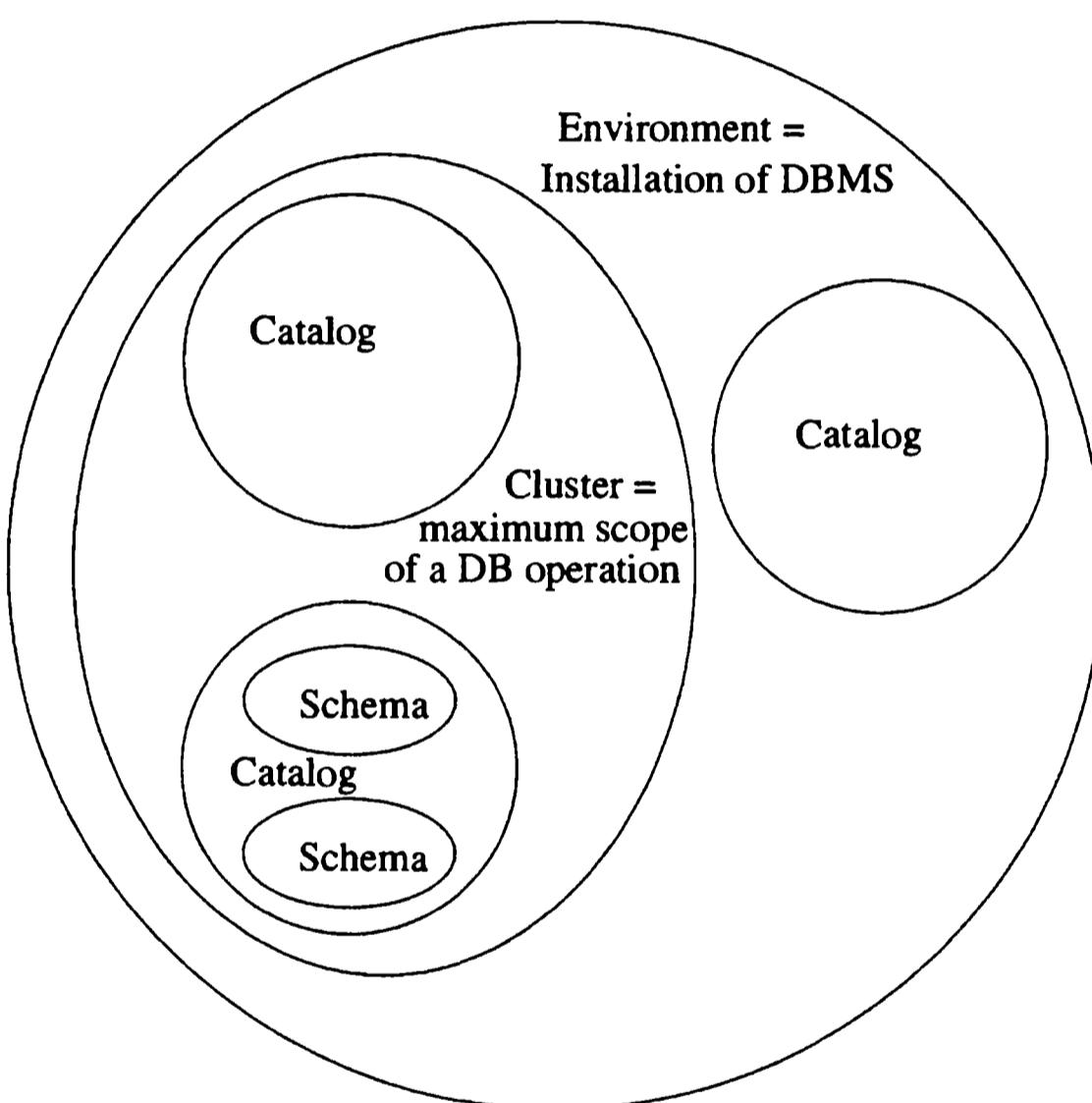


Figure 9.2: Organization of database elements within the environment

Briefly, the organization consists of the following structures:

1. *Schemas*. These are collections of tables, views, assertions, triggers, and some other types of information (see the box on “More Schema Elements” in Section 9.2.2). Schemas are the basic units of organization, close to what we might think of as a “database,” but in fact somewhat less than a database as we shall see in point (3) below.
2. *Catalogs*. These are collections of schemas. They are the basic unit for supporting unique, accessible terminology. Each catalog has one or more schemas; the names of schemas within a catalog must be unique, and

each catalog contains a special schema called **INFORMATION\_SCHEMA** that contains information about all the schemas in the catalog.

3. *Clusters.* These are collections of catalogs. Each user has an associated cluster: the set of all catalogs accessible to the user (see Section 10.1 for an explanation of how access to catalogs and other elements is controlled). A cluster is the maximum scope over which a query can be issued, so in a sense, a cluster is “the database” as seen by a particular user.

### 9.2.2 Schemas

The simplest form of schema declaration is:

```
CREATE SCHEMA <schema name> <element declarations>
```

The element declarations are of the forms discussed in various places, such as Sections 2.3, 8.1.1, 7.5.1, and 9.4.1.

**Example 9.2 :** We could declare a schema that includes the five relations about movies that we have been using in our running example, plus some of the other elements we have introduced, such as views. Figure 9.3 sketches the form of such a declaration. □

```
CREATE SCHEMA MovieSchema
  CREATE TABLE MovieStar ... as in Fig. 7.3
    Create-table statements for the four other tables
  CREATE VIEW MovieProd ... as in Example 8.2
    Other view declarations
  CREATE ASSERTION RichPres ... as in Example 7.11
```

Figure 9.3: Declaring a schema

It is not necessary to declare the schema all at once. One can modify or add to the “current” schema using the appropriate **CREATE**, **DROP**, or **ALTER** statement, e.g., **CREATE TABLE** followed by the declaration of a new table for the schema. We change the “current” schema with a **SET SCHEMA** statement. For example,

```
SET SCHEMA MovieSchema;
```

makes the schema described in Fig. 9.3 the current schema. Then, any declarations of schema elements are added to that schema, and any **DROP** or **ALTER** statements refer to elements already in that schema.

## More Schema Elements

Some schema elements that we have not already mentioned, but that occasionally are useful are:

- *Domains*: These are sets of values or simple data types. They are little used today, because object-relational DBMS's provide more powerful type-creation mechanisms; see Section 10.4.
- *Character sets*: These are sets of symbols and methods for encoding them. ASCII and Unicode are common options.
- *Collations*: A collation specifies which characters are “less than” which others. For example, we might use the ordering implied by the ASCII code, or we might treat lower-case and capital letters the same and not compare anything that isn’t a letter.
- *Grant statements*: These concern who has access to schema elements. We shall discuss the granting of privileges in Section 10.1.
- *Stored Procedures*: These are executable code; see Section 9.4.

### 9.2.3 Catalogs

Just as schema elements like tables are created within a schema, schemas are created and modified within a catalog. In principle, we would expect the process of creating and populating catalogs to be analogous to the process of creating and populating schemas. Unfortunately, SQL does not define a standard way to do so, such as a statement

`CREATE CATALOG <catalog name>`

followed by a list of schemas belonging to that catalog and the declarations of those schemas.

However, SQL does stipulate a statement

`SET CATALOG <catalog name>`

This statement allows us to set the “current” catalog, so new schemas will go into that catalog and schema modifications will refer to schemas in that catalog should there be a name ambiguity.

### 9.2.4 Clients and Servers in the SQL Environment

A SQL environment is more than a collection of catalogs and schemas. It contains elements whose purpose is to support operations on the database or

## Complete Names for Schema Elements

Formally, the name for a schema element such as a table is its catalog name, its schema name, and its own name, connected by dots in that order. Thus, the table **Movies** in the schema **MovieSchema** in the catalog **MovieCatalog** can be referred to as

**MovieCatalog.MovieSchema.Movies**

If the catalog is the default or current catalog, then we can omit that component of the name. If the schema is also the default or current schema, then that part too can be omitted, and we are left with the element's own name, as is usual. However, we have the option to use the full name if we need to access something outside the current schema or catalog.

databases represented by those catalogs and schemas. According to the SQL standard, within a SQL environment are two special kinds of processes: SQL clients and SQL servers.

In terms of Fig. 9.1, a “SQL server” plays the role what we called a “database server there. A “SQL client” is like the application servers from that figure. The SQL standard does not define processes analogous to what we called “Web servers” or “clients” in Fig. 9.1.

### 9.2.5 Connections

If we wish to run some program involving SQL at a host where a SQL client exists, then we may open a connection between the client and server by executing a SQL statement

```
CONNECT TO <server name> AS <connection name>
    AUTHORIZATION <name and password>
```

The server name is something that depends on the installation. The word **DEFAULT** can substitute for a name and will connect the user to whatever SQL server the installation treats as the “default server.” We have shown an authorization clause followed by the user’s name and password. The latter is the typical method by which a user would be identified to the server, although other strings following **AUTHORIZATION** might be used.

The connection name can be used to refer to the connection later on. The reason we might have to refer to the connection is that SQL allows several connections to be opened by the user, but only one can be active at any time. To switch among connections, we can make **conn1** become the active connection by the statement:

```
SET CONNECTION conn1;
```

Whatever connection was currently active becomes *dormant* until it is reactivated with another **SET CONNECTION** statement that mentions it explicitly.

We also use the name when we drop the connection. We can drop connection **conn1** by

```
DISCONNECT conn1;
```

Now, **conn1** is terminated; it is not dormant and cannot be reactivated.

However, if we shall never need to refer to the connection being created, then **AS** and the connection name may be omitted from the **CONNECT TO** statement. It is also permitted to skip the connection statements altogether. If we simply execute SQL statements at a host with a SQL client, then a default connection will be established on our behalf.

### 9.2.6 Sessions

The SQL operations that are performed while a connection is active form a *session*. The session lasts as long as the connection that created it. For example, when a connection is made dormant, its session also becomes dormant, and reactivation of the connection by a **SET CONNECTION** statement also makes the session active. Thus, we have shown the session and connection as two aspects of the link between client and server in Fig. 9.4.

Each session has a current catalog and a current schema within that catalog. These may be set with statements **SET SCHEMA** and **SET CATALOG**, as discussed in Sections 9.2.2 and 9.2.3. There is also an authorized user for every session, as we shall discuss in Section 10.1.

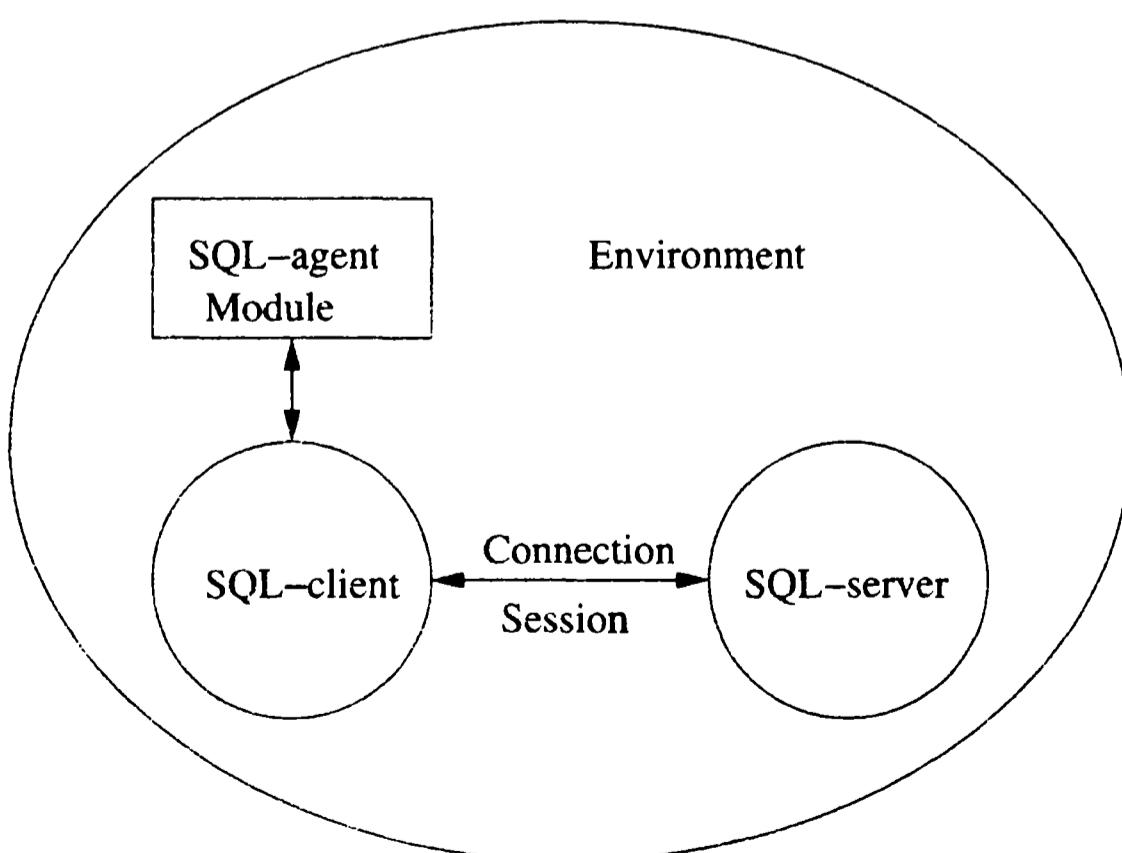


Figure 9.4: The SQL client-server interactions

## The Languages of the SQL Standard

Implementations conforming to the SQL standard are required to support at least one of the following seven host languages: ADA, C, Cobol, Fortran, M (formerly called Mumps, and used primarily in the medical community), Pascal, and PL/I. We shall use C in our examples.

### 9.2.7 Modules

A *module* is the SQL term for an application program. The SQL standard suggests that there are three kinds of modules, but insists only that a SQL implementation offer the user at least one of these types.

1. *Generic SQL Interface.* The user may type SQL statements that are executed by a SQL server. In this mode, each query or other statement is a module by itself. It is this mode that we imagined for most of our examples in this book, although in practice it is rarely used.
2. *Embedded SQL.* This style will be discussed in Section 9.3. Typically, a preprocessor turns the embedded SQL statements into suitable function or procedure calls to the SQL system. The compiled host-language program, including these function calls, is a module.
3. *True Modules.* The most general style of modules envisioned by SQL is a collection of stored functions or procedures, some of which are host-language code and some of which are SQL statements. They communicate among themselves by passing parameters and perhaps via shared variables. PSM modules (Section 9.4) are an example of this type of module.

An execution of a module is called a *SQL agent*. In Fig. 9.4 we have shown both a module and an SQL agent, as one unit, calling upon a SQL client to establish a connection. However, we should remember that the distinction between a module and an SQL agent is analogous to the distinction between a program and a process; the first is code, the second is an execution of that code.

## 9.3 The SQL/Host-Language Interface

To this point, we have used the *generic SQL interface* in our examples. That is, we have assumed there is a SQL interpreter, which accepts and executes the sorts of SQL queries and commands that we have learned. Although provided as an option by almost all DBMS's, this mode of operation is actually rare. In real systems, such as those described in Section 9.1, there is a program in some

conventional *host* language such as C, but some of the steps in this program are actually SQL statements.

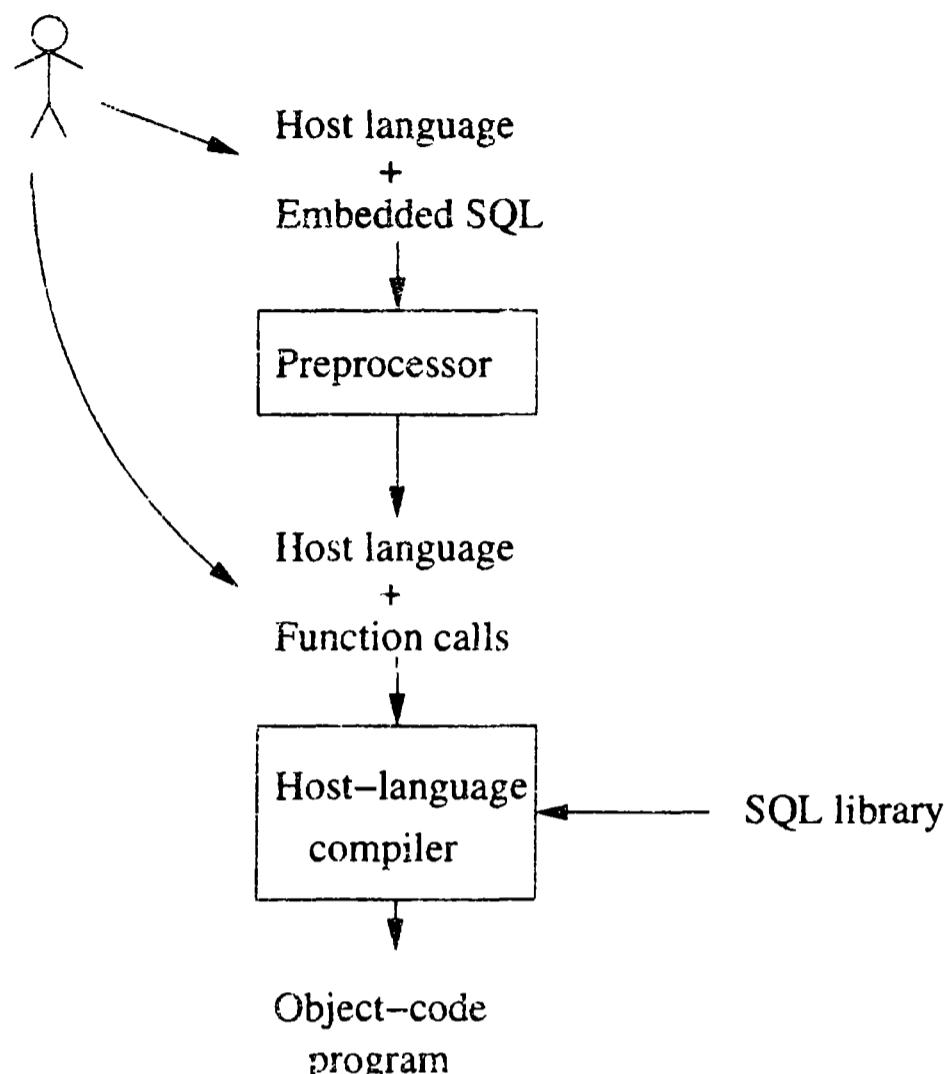


Figure 9.5: Processing programs with SQL statements embedded

A sketch of a typical programming system that involves SQL statements is in Fig. 9.5. There, we see the programmer writing programs in a host language, but with some special “embedded” SQL statements. There are two ways this embedding could take place.

1. *Call-Level Interface.* A library is provided, and the embedding of SQL in the host language is really calls to functions or methods in this library. SQL statements are usually string arguments of these methods. This approach, often referred to as a call-level interface or CLI, is discussed in Section 9.5 and is represented by the curved arrow in Fig. 9.5 from the user directly to the host language.
2. *Directly Embedded SQL.* The entire host-language program, with embedded SQL statements, is sent to a preprocessor, which changes the embedded SQL statements into something that makes sense in the host language. Typically, the SQL statements are replaced by calls to library functions or methods, so the difference between a CLI and direct embedding of SQL is more a matter of “look and feel” than of substance. The preprocessed host-language program is then compiled in the usual manner and operates on the database through execution of the library calls.

In this section, we shall learn the SQL standard for direct embedding in a host language — C in particular. We are also introduced to a number of concepts, such as cursors, that appear in all, or almost all, systems for embedding SQL.

### 9.3.1 The Impedance Mismatch Problem

The basic problem of connecting SQL statements with those of a conventional programming language is *impedance mismatch*: the fact that the data model of SQL differs so much from the models of other languages. As we know, SQL uses the relational data model at its core. However, C and similar languages use a data model with integers, reals, arithmetic, characters, pointers, record structures, arrays, and so on. Sets are not represented directly in C or these other languages, while SQL does not use pointers, loops and branches, or many other common programming-language constructs. As a result, passing data between SQL and other languages is not straightforward, and a mechanism must be devised to allow the development of programs that use both SQL and another language.

One might first suppose that it is preferable to use a single language. Either do all computation in SQL or forget SQL and do all computation in a conventional language. However, we can dispense with the idea of omitting SQL when there are database operations involved. SQL systems greatly aid the programmer in writing database operations that can be executed efficiently, yet that can be expressed at a very high level. SQL takes from the programmer's shoulders the need to understand how data is organized in storage or how to exploit that storage structure to operate efficiently on the database.

On the other hand, there are many important things that SQL cannot do at all. For example, one cannot write a SQL query to compute  $n$  factorial, something that is an easy exercise in C or similar languages.<sup>1</sup> As another example, SQL cannot format its output directly into a convenient form such as a graphic. Thus, real database programming requires both SQL and a host language.

### 9.3.2 Connecting SQL to the Host Language

When we wish to use a SQL statement within a host-language program, we warn the preprocessor that SQL code is coming with the keywords `EXEC SQL` in front of the statement. We transfer information between the database, which is accessed only by SQL statements, and the host-language program through *shared variables*, which are allowed to appear in both host-language statements

---

<sup>1</sup>We should be careful here. There are extensions to the basic SQL language, such as recursive SQL discussed in Section 10.2 or SQL/PSM discussed in Section 9.4, that do offer “Turing completeness” — the ability to compute anything that can be computed in any other programming language. However, these extensions were never intended for general-purpose calculation, and we do not regard them as general-purpose languages.

and SQL statements. Shared variables are prefixed by a colon within a SQL statement, but they appear without the colon in host-language statements.

A special variable, called **SQLSTATE** in the SQL standard, serves to connect the host-language program with the SQL execution system. The type of **SQLSTATE** is an array of five characters. Each time a function of the SQL library is called, a code is put in the variable **SQLSTATE** that indicates any problems found during that call. The SQL standard also specifies a large number of five-character codes and their meanings.

For example, '00000' (five zeroes) indicates that no error condition occurred, and '02000' indicates that a tuple requested as part of the answer to a SQL query could not be found. The latter code is very important, since it allows us to create a loop in the host-language program that examines tuples from some relation one-at-a-time and to break the loop after the last tuple has been examined.

### 9.3.3 The DECLARE Section

To declare shared variables, we place their declarations between two embedded SQL statements:

```
EXEC SQL BEGIN DECLARE SECTION;
...
EXEC SQL END DECLARE SECTION;
```

What appears between them is called the *declare section*. The form of variable declarations in the declare section is whatever the host language requires. It only makes sense to declare variables to have types that both the host language and SQL can deal with, such as integers, reals, and character strings or arrays.

**Example 9.3:** The following statements might appear in a C function that updates the **Studio** relation:

```
EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50], studioAddr[256];
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

The first and last statements are the required beginning and end of the declare section. In the middle is a statement declaring two shared variables, **studioName** and **studioAddr**. These are both character arrays and, as we shall see, they can be used to hold a name and address of a studio that are made into a tuple and inserted into the **Studio** relation. The third statement declares **SQLSTATE** to be a six-character array.<sup>2</sup> □

---

<sup>2</sup>We shall use six characters for the five-character value of **SQLSTATE** because in programs to follow we want to use the C function **strcmp** to test whether **SQLSTATE** has a certain value. Since **strcmp** expects strings to be terminated by '\0', we need a sixth character for this endmarker. The sixth character must be set initially to '\0', but we shall not show this assignment in programs to follow.

### 9.3.4 Using Shared Variables

A shared variable can be used in SQL statements in places where we expect or allow a constant. Recall that shared variables are preceded by a colon when so used. Here is an example in which we use the variables of Example 9.3 as components of a tuple to be inserted into relation `Studio`.

**Example 9.4:** In Fig. 9.6 is a sketch of a C function `getStudio` that prompts the user for the name and address of a studio, reads the responses, and inserts the appropriate tuple into `Studio`. Lines (1) through (4) are the declarations from Example 9.3. We omit the C code that prints requests and scans entered text to fill the two arrays `studioName` and `studioAddr`.

```

void getStudio() {

1)      EXEC SQL BEGIN DECLARE SECTION;
2)          char studioName[50], studioAddr[256];
3)          char SQLSTATE[6];
4)      EXEC SQL END DECLARE SECTION;

    /* print request that studio name and address
       be entered and read response into variables
       studioName and studioAddr */

5)      EXEC SQL INSERT INTO Studio(name, address)
6)          VALUES (:studioName, :studioAddr);
}

```

Figure 9.6: Using shared variables to insert a new studio

Then, in lines (5) and (6) is an embedded SQL `INSERT` statement. This statement is preceded by the keywords `EXEC SQL` to indicate that it is indeed an embedded SQL statement rather than ungrammatical C code. The values inserted by lines (5) and (6) are not explicit constants, as they were in all previous examples; rather, the values appearing in line (6) are shared variables whose current values become components of the inserted tuple. □

Any SQL statement that does not return a result (i.e., is not a query) can be embedded in a host-language program by preceding it with `EXEC SQL`. Examples of embeddable SQL statements include insert-, delete-, and update-statements and those statements that create, modify, or drop schema elements such as tables and views.

However, select-from-where queries are not embeddable directly into a host language, because of the “impedance mismatch.” Queries produce bags of tuples as a result, while none of the major host languages support a set or bag

data type directly. Thus, embedded SQL must use one of two mechanisms for connecting the result of queries with a host-language program:

1. *Single-Row SELECT Statements.* A query that produces a single tuple can have that tuple stored in shared variables, one variable for each component of the tuple.
2. *Cursors.* Queries producing more than one tuple can be executed if we declare a *cursor* for the query. The cursor ranges over all tuples in the answer relation, and each tuple in turn can be fetched into shared variables and processed by the host-language program.

We shall consider each of these mechanisms in turn.

### 9.3.5 Single-Row Select Statements

The form of a single-row select is the same as an ordinary select-from-where statement, except that following the **SELECT** clause is the keyword **INTO** and a list of shared variables. These shared variables each are preceded by a colon, as is the case for all shared variables within a SQL statement. If the result of the query is a single tuple, this tuple's components become the values of these variables. If the result is either no tuple or more than one tuple, then no assignment to the shared variables is made, and an appropriate error code is written in the variable **SQLSTATE**.

**Example 9.5 :** We shall write a C function to read the name of a studio and print the net worth of the studio's president. A sketch of this function is shown in Fig. 9.7. It begins with a declare section, lines (1) through (5), for the variables we shall need. Next, C statements that we do not show explicitly obtain a studio name from the standard input.

Lines (6) through (9) are the single-row select statement. It is quite similar to queries we have already seen. The two differences are that the value of variable **studioName** is used in place of a constant string in the condition of line (9), and there is an **INTO** clause at line (7) that tells us where to put the result of the query. In this case, we expect a single tuple, and tuples have only one component, that for attribute **netWorth**. The value of this one component of one tuple is placed in the shared variable **presNetWorth**. □

### 9.3.6 Cursors

The most versatile way to connect SQL queries to a host language is with a cursor that runs through the tuples of a relation. This relation can be a stored table, or it can be something that is generated by a query. To create and use a cursor, we need the following statements:

1. A cursor declaration, whose simplest form is:

```

void printNetWorth() {

1)      EXEC SQL BEGIN DECLARE SECTION;
2)          char studioName[50];
3)          int presNetWorth;
4)          char SQLSTATE[6];
5)      EXEC SQL END DECLARE SECTION;

/* print request that studio name be entered.
   read response into studioName */

6)      EXEC SQL SELECT netWorth
7)          INTO :presNetWorth
8)          FROM Studio, MovieExec
9)          WHERE presC# = cert# AND
               Studio.name = :studioName;

/* check that SQLSTATE has all 0's and if so, print
   the value of presNetWorth */
}

```

Figure 9.7: A single-row select embedded in a C function

**EXEC SQL DECLARE <cursor name> CURSOR FOR <query>**

The query can be either an ordinary select-from-where query or a relation name. The cursor *ranges* over the tuples of the relation produced by the query.

2. A statement **EXEC SQL OPEN**, followed by the cursor name. This statement initializes the cursor to a position where it is ready to retrieve the first tuple of the relation over which the cursor ranges.
3. One or more uses of a *fetch statement*. The purpose of a fetch statement is to get the next tuple of the relation over which the cursor ranges. The fetch statement has the form:

**EXEC SQL FETCH FROM <cursor name> INTO <list of variables>**

There is one variable in the list for each attribute of the tuple's relation. If there is a tuple available to be fetched, these variables are assigned the values of the corresponding components from that tuple. If the tuples have been exhausted, then no tuple is returned, and the value of SQLSTATE is set to '02000', a code that means "no tuple found."

4. The statement `EXEC SQL CLOSE` followed by the name of the cursor. This statement closes the cursor, which now no longer ranges over tuples of the relation. It can, however, be reinitialized by another `OPEN` statement, in which case it ranges anew over the tuples of this relation.

**Example 9.6:** Suppose we wish to determine the number of movie executives whose net worths fall into a sequence of bands of exponentially growing size, each band corresponding to a number of digits in the net worth. We shall design a query that retrieves the `netWorth` field of all the `MovieExec` tuples into a shared variable called `worth`. A cursor called `execCursor` will range over all these one-component tuples. Each time a tuple is fetched, we compute the number of digits in the integer `worth` and increment the appropriate element of an array `counts`.

The C function `worthRanges` begins in line (1) of Fig. 9.8. Line (2) declares some variables used only by the C function, not by the embedded SQL. The array `counts` holds the counts of executives in the various bands, `digits` counts the number of digits in a net worth, and `i` is an index ranging over the elements of array `counts`.

Lines (3) through (6) are a SQL declare section in which shared variable `worth` and the usual `SQLSTATE` are declared. Lines (7) and (8) declare `execCursor` to be a cursor that ranges over the values produced by the query on line (8). This query simply asks for the `netWorth` components of all the tuples in `MovieExec`. This cursor is then opened at line (9). Line (10) completes the initialization by zeroing the elements of array `counts`.

The main work is done by the loop of lines (11) through (16). At line (12) a tuple is fetched into shared variable `worth`. Since tuples produced by the query of line (8) have only one component, we need only one shared variable, although in general there would be as many variables as there are components of the retrieved tuples. Line (13) tests whether the fetch has been successful. Here, we use a macro `NO_MORE_TUPLES`, defined by

```
#define NO_MORE_TUPLES !(strcmp(SQLSTATE,"02000"))
```

Recall that "02000" is the `SQLSTATE` code that means no tuple was found. If there are no more tuples, we break out of the loop and go to line (17).

If a tuple has been fetched, then at line (14) we initialize the number of digits in the net worth to 1. Line (15) is a loop that repeatedly divides the net worth by 10 and increments `digits` by 1. When the net worth reaches 0 after division by 10, `digits` holds the correct number of digits in the value of `worth` that was originally retrieved. Finally, line (16) increments the appropriate element of the array `counts` by 1. We assume that the number of digits is no more than 14. However, should there be a net worth with 15 or more digits, line (16) will not increment any element of the `counts` array, since there is no appropriate range; i.e., enormous net worths are thrown away and do not affect the statistics.

Line (17) begins the wrap-up of the function. The cursor is closed, and lines (18) and (19) print the values in the `counts` array. □

```

1) void worthRanges() {

2)     int i, digits, counts[15];
3)     EXEC SQL BEGIN DECLARE SECTION;
4)         int worth;
5)         char SQLSTATE[6];
6)     EXEC SQL END DECLARE SECTION;
7)     EXEC SQL DECLARE execCursor CURSOR FOR
8)         SELECT netWorth FROM MovieExec;

9)     EXEC SQL OPEN execCursor;
10)    for(i=1; i<15; i++) counts[i] = 0;
11)    while(1) {
12)        EXEC SQL FETCH FROM execCursor INTO :worth;
13)        if(NO_MORE_TUPLES) break;
14)        digits = 1;
15)        while((worth /= 10) > 0) digits++;
16)        if(digits <= 14) counts[digits]++;
17)    }
18)    EXEC SQL CLOSE execCursor;
19)    for(i=0; i<15; i++)
20)        printf("digits = %d: number of execs = %d\n",
           i, counts[i]);
}

```

Figure 9.8: Grouping executive net worths into exponential bands

### 9.3.7 Modifications by Cursor

When a cursor ranges over the tuples of a base table (i.e., a relation that is stored in the database), then one can not only read the current tuple, but one can update or delete the current tuple. The syntax of these UPDATE and DELETE statements are the same as we encountered in Section 6.5, with the exception of the WHERE clause. That clause may only be WHERE CURRENT OF followed by the name of the cursor. Of course it is possible for the host-language program reading the tuple to apply whatever condition it likes to the tuple before deciding whether or not to delete or update it.

**Example 9.7:** In Fig. 9.9 we see a C function that looks at each tuple of `MovieExec` and decides either to delete the tuple or to double the net worth. In lines (3) and (4) we declare variables that correspond to the four attributes of `MovieExec`, as well as the necessary `SQLSTATE`. Then, at line (6), `execCursor` is declared to range over the stored relation `MovieExec` itself.

Lines (8) through (14) are the loop, in which the cursor `execCursor` refers to each tuple of `MovieExec`, in turn. Line (9) fetches the current tuple into

```

1) void changeWorth() {

2)     EXEC SQL BEGIN DECLARE SECTION;
3)         int certNo, worth;
4)         char execName[31], execAddr[256], SQLSTATE[6];
5)     EXEC SQL END DECLARE SECTION;
6)     EXEC SQL DECLARE execCursor CURSOR FOR MovieExec;

7)     EXEC SQL OPEN execCursor;
8)     while(1) {
9)         EXEC SQL FETCH FROM execCursor INTO :execName,
10)             :execAddr, :certNo, :worth;
11)         if(NO_MORE_TUPLES) break;
12)         if (worth < 1000)
13)             EXEC SQL DELETE FROM MovieExec
14)                 WHERE CURRENT OF execCursor;
15)         else
16)             EXEC SQL UPDATE MovieExec
17)                 SET netWorth = 2 * netWorth
18)                 WHERE CURRENT OF execCursor;
19)     }
20)     EXEC SQL CLOSE execCursor;
}

```

Figure 9.9: Modifying executive net worths

the four variables used for this purpose; note that only `worth` is actually used. Line (10) tests whether we have exhausted the tuples of `MovieExec`. We have again used the macro `NO_MORE_TUPLES` for the condition that variable `SQLSTATE` has the “no more tuples” code “02000”.

In the test of line (11) we ask if the net worth is under \$1000. If so, the tuple is deleted by the `DELETE` statement of line (12). Note that the `WHERE` clause refers to the cursor, so the current tuple of `MovieExec`, the one we just fetched, is deleted from `MovieExec`. If the net worth is at least \$1000, then at line (14), the net worth in the same tuple is doubled, instead. □

### 9.3.8 Protecting Against Concurrent Updates

Suppose that as we examine the net worths of movie executives using the function `worthRanges` of Fig. 9.8, some other process is modifying the underlying `MovieExec` relation. What should we do about this possibility? Perhaps nothing. We might be happy with approximate statistics, and we don’t care whether or not we count an executive who was in the process of being deleted, for example. Then, we simply accept what tuples we get through the cursor.

However, we may not wish to allow concurrent changes to affect the tuples we see through this cursor. Rather, we may insist on the statistics being taken on the relation as it exists at some point in time. In terms of the transactions of Section 6.6, we want the code that runs the cursor through the relation to be serializable with any other operations on the relation. To obtain this guarantee, we may declare the cursor *insensitive* to concurrent changes.

**Example 9.8:** We could modify lines (7) and (8) of Fig. 9.8 to be:

```
7)      EXEC SQL DECLARE execCursor INSENSITIVE CURSOR FOR
8)          SELECT netWorth FROM MovieExec;
```

If `execCursor` is so declared, then the SQL system will guarantee that changes to relation `MovieExec` made between one opening and closing of `execCursor` will not affect the set of tuples fetched. □

There are certain cursors ranging over a relation  $R$  about which we may say with certainty that they will not change  $R$ . Such a cursor can run simultaneously with an insensitive cursor for  $R$ , without risk of changing the relation  $R$  that the insensitive cursor sees. If we declare a cursor `FOR READ ONLY`, then the database system can be sure that the underlying relation will not be modified because of access to the relation through this cursor.

**Example 9.9:** We could append after line (8) of `worthRanges` in Fig. 9.8 a line

```
FOR READ ONLY;
```

If so, then any attempt to execute a modification through cursor `execCursor` would cause an error. □

### 9.3.9 Dynamic SQL

Our model of SQL embedded in a host language has been that of specific SQL queries and commands within a host-language program. An alternative style of embedded SQL has the statements themselves be computed by the host language. Such statements are not known at compile time, and thus cannot be handled by a SQL preprocessor or a host-language compiler.

An example of such a situation is a program that prompts the user for an SQL query, reads the query, and then executes that query. The generic interface for ad-hoc SQL queries that we assumed in Chapter 6 is an example of just such a program. If queries are read and executed at run-time, there is nothing that can be done at compile-time. The query has to be parsed and a suitable way to execute the query found by the SQL system, immediately after the query is read.

The host-language program must instruct the SQL system to take the character string just read, to turn it into an executable SQL statement, and finally to execute that statement. There are two *dynamic SQL* statements that perform these two steps.

1. **EXEC SQL PREPARE  $V$  FROM <expression>**, where  $V$  is a SQL variable. The expression can be any host-language expression whose value is a string; this string is treated as a SQL statement. Presumably, the SQL statement is parsed and a good way to execute it is found by the SQL system, but the statement is not executed. Rather, the plan for executing the SQL statement becomes the value of  $V$ .
2. **EXEC SQL EXECUTE  $V$** . This statement causes the SQL statement denoted by variable  $V$  to be executed.

Both steps can be combined into one, with the statement:

**EXEC SQL EXECUTE IMMEDIATE <expression>**

The disadvantage of combining these two parts is seen if we prepare a statement once and then execute it many times. With **EXECUTE IMMEDIATE** the cost of preparing the statement is paid each time the statement is executed, rather than paid only once, when we prepare it.

**Example 9.10:** In Fig. 9.10 is a sketch of a C program that reads text from standard input into a variable **query**, prepares it, and executes it. The SQL variable **SQLquery** holds the prepared query. Since the query is only executed once, the line:

```
EXEC SQL EXECUTE IMMEDIATE :query;
```

could replace lines (6) and (7) of Fig. 9.10. □

```

1) void readQuery() {
2)     EXEC SQL BEGIN DECLARE SECTION;
3)         char *query;
4)     EXEC SQL END DECLARE SECTION;
5)     /* prompt user for a query, allocate space (e.g.,
       use malloc) and make shared variable :query point
       to the first character of the query */
6)     EXEC SQL PREPARE SQLquery FROM :query;
7)     EXEC SQL EXECUTE SQLquery;
}
```

Figure 9.10: Preparing and executing a dynamic SQL query

### 9.3.10 Exercises for Section 9.3

**Exercise 9.3.1:** Write the following embedded SQL queries, based on the database schema

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

of Exercise 2.4.1. You may use any host language with which you are familiar, and details of host-language programming may be replaced by clear comments if you wish.

- a) Ask the user for a price and find the PC whose price is closest to the desired price. Print the maker, model number, and speed of the PC.
- b) Ask the user for minimum values of the speed, RAM, hard-disk size, and screen size that they will accept. Find all the laptops that satisfy these requirements. Print their specifications (all attributes of `laptop`) and their manufacturer.
- ! c) Ask the user for a manufacturer. Print the specifications of all products by that manufacturer. That is, print the model number, product-type, and all the attributes of whichever relation is appropriate for that type.
- !! d) Ask the user for a “budget” (total price of a PC and printer), and a minimum speed of the PC. Find the cheapest “system” (PC plus printer) that is within the budget and minimum speed, but make the printer a color printer if possible. Print the model numbers for the chosen system.
- e) Ask the user for a manufacturer, model number, speed, RAM, hard-disk size, and price of a new PC. Check that there is no PC with that model number. Print a warning if so, and otherwise insert the information into tables `Product` and `PC`.

**Exercise 9.3.2:** Write the following embedded SQL queries, based on the database schema

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

of Exercise 2.4.3.

- a) The firepower of a ship is roughly proportional to the number of guns times the cube of the bore of the guns. Find the class with the largest firepower.

- ! b) Ask the user for the name of a battle. Find the countries of the ships involved in the battle. Print the country with the most ships sunk and the country with the most ships damaged.
- c) Ask the user for the name of a class and the other information required for a tuple of table **Classes**. Then ask for a list of the names of the ships of that class and their dates launched. However, the user need not give the first name, which will be the name of the class. Insert the information gathered into **Classes** and **Ships**.
- ! d) Examine the **Battles**, **Outcomes**, and **Ships** relations for ships that were in battle before they were launched. Prompt the user when there is an error found, offering the option to change the date of launch or the date of the battle. Make whichever change is requested.

## 9.4 Stored Procedures

In this section, we introduce you to *Persistent, Stored Modules* (SQL/PSM, or just PSM). PSM is part of the latest revision to the SQL standard, called SQL:2003. It allows us to write procedures in a simple, general-purpose language and to store them in the database, as part of the schema. We can then use these procedures in SQL queries and other statements to perform computations that cannot be done with SQL alone. Each commercial DBMS offers its own extension of PSM. In this book, we shall describe the SQL/PSM standard, which captures the major ideas of these facilities, and which should help you understand the language associated with any particular system. References to PSM extensions provided with several major commercial systems are in the bibliographic notes.

### 9.4.1 Creating PSM Functions and Procedures

In PSM, you define *modules*, which are collections of function and procedure definitions, temporary relation declarations, and several other optional declarations. The major elements of a procedure declaration are:

```
CREATE PROCEDURE <name> (<parameters>)
    <local declarations>
    <procedure body>;
```

This form should be familiar from a number of programming languages; it consists of a procedure name, a parenthesized list of parameters, some optional local-variable declarations, and the executable body of code that defines the procedure. A function is defined in almost the same way, except that the keyword **FUNCTION** is used, and there is a return-value type that must be specified. That is, the elements of a function definition are:

```
CREATE FUNCTION <name> (<parameters>) RETURNS <type>
  <local declarations>
  <function body>;
```

The parameters of a PSM procedure are mode-name-type triples. That is, the parameter name is not only followed by its declared type, as usual in programming languages, but it is preceded by a “mode,” which is either `IN`, `OUT`, or `INOUT`. These three keywords indicate that the parameter is input-only, output-only, or both input and output, respectively. `IN` is the default, and can be omitted.

Function parameters, on the other hand, may only be of mode `IN`. That is, PSM forbids side-effects in functions, so the only way to obtain information from a function is through its return-value. We shall not specify the `IN` mode for function parameters, although we do so in procedure definitions.

**Example 9.11:** While we have not yet learned the variety of statements that can appear in procedure and function bodies, one kind should not surprise us: an SQL statement. The limitation on these statements is the same as for embedded SQL, as we introduced in Section 9.3.4: only single-row-select statements and cursor-based accesses are permitted as queries. In Fig. 9.11 is a PSM procedure that takes two addresses — an old address and a new address — as parameters and replaces the old address by the new everywhere it appears in `MovieStar`.

```
1) CREATE PROCEDURE Move(
2)   IN oldAddr VARCHAR(255),
3)   IN newAddr VARCHAR(255)
4) )
5) UPDATE MovieStar
6) SET address = newAddr
7) WHERE address = oldAddr;
```

Figure 9.11: A procedure to change addresses

Line (1) introduces the procedure and its name, `Move`. Lines (2) and (3) declare two input parameters, both of whose types are `VARCHAR(255)`. This type is consistent with the type we declared for the attribute `address` of `MovieStar` in Fig. 2.8. Lines (4) through (6) are a conventional `UPDATE` statement. However, notice that the parameter names can be used as if they were constants. Unlike host-language variables, which require a colon prefix when used in SQL (see Section 9.3.2), parameters and other local variables of PSM procedures and functions require no colon. □

## 9.4.2 Some Simple Statement Forms in PSM

Let us begin with a potpourri of statement forms that are easy to master.

1. *The call-statement:* The form of a procedure call is:

```
CALL <procedure name> (<argument list>);
```

That is, the keyword **CALL** is followed by the name of the procedure and a parenthesized list of arguments, as in most any language. This call can, however, be made from a variety of places:

- i. From a host-language program, in which it might appear as

```
EXEC SQL CALL Foo(:x, 3);
```

for instance.

- ii. As a statement of another PSM function or procedure.
  - iii. As a SQL command issued to the generic SQL interface. For example, we can issue a statement such as

```
CALL Foo(1, 3);
```

to such an interface, and have stored procedure **Foo** executed with its two parameters set equal to 1 and 3, respectively.

Note that it is not permitted to call a function. You invoke functions in PSM as you do in C: use the function name and suitable arguments as part of an expression.

2. *The return-statement:* Its form is

```
RETURN <expression>;
```

This statement can only appear in a function. It evaluates the expression and sets the return-value of the function equal to that result. However, at variance with common programming languages, the return-statement of PSM does *not* terminate the function. Rather, control continues with the following statement, and it is possible that the return-value will be changed before the function completes.

3. *Declarations of local variables:* The statement form

```
DECLARE <name> <type>;
```

declares a variable with the given name to have the given type. This variable is local, and its value is not preserved by the DBMS after a running of the function or procedure. Declarations must precede executable statements in the function or procedure body.

4. *Assignment Statements:* The form of an assignment is:

`SET <variable> = <expression>;`

Except for the introductory keyword `SET`, assignment in PSM is quite like assignment in other languages. The expression on the right of the equal-sign is evaluated, and its value becomes the value of the variable on the left. `NULL` is a permissible expression. The expression may even be a query, as long as it returns a single value.

5. *Statement groups*: We can form a list of statements ended by semicolons and surrounded by keywords `BEGIN` and `END`. This construct is treated as a single statement and can appear anywhere a single statement can. In particular, since a procedure or function body is expected to be a single statement, we can put any sequence of statements in the body by surrounding them by `BEGIN...END`.
6. *Statement labels*: We label a statement by prefixing it with a name (the label) and a colon.

### 9.4.3 Branching Statements

For our first complex PSM statement type, let us consider the if-statement. The form is only a little strange; it differs from C or similar languages in that:

1. The statement ends with keywords `END IF`.
2. If-statements nested within the else-clause are introduced with the single word `ELSEIF`.

Thus, the general form of an if-statement is as suggested by Fig. 9.12. The condition is any boolean-valued expression, as can appear in the `WHERE` clause of SQL statements. Each statement list consists of statements ended by semicolons, but does not need a surrounding `BEGIN...END`. The final `ELSE` and its statement(s) are optional; i.e., `IF...THEN...END IF` alone or with `ELSEIF`'s is acceptable.

**Example 9.12:** Let us write a function to take a year  $y$  and a studio  $s$ , and return a boolean that is `TRUE` if and only if studio  $s$  produced at least one comedy in year  $y$  or did not produce any movies at all in that year. The code appears in Fig. 9.13.

Line (1) introduces the function and includes its arguments. We do not need to specify a mode for the arguments, since that can only be `IN` for a function. Lines (2) and (3) test for the case where there are no movies at all by studio  $s$  in year  $y$ , in which case we set the return-value to `TRUE` at line (4). Note that line (4) does not cause the function to return. Technically, it is the flow of control dictated by the if-statements that causes control to jump from line (4) to line (9), where the function completes and returns.

```

IF <condition> /tt THEN
    <statement list>
ELSEIF <condition> /tt THEN
    <statement list>
ELSEIF
    ...
ELSE
    <statement list>
END IF;

```

Figure 9.12: The form of an if-statement

```

1) CREATE FUNCTION BandW(y INT, s CHAR(15)) RETURNS BOOLEAN
2) IF NOT EXISTS(
3)     SELECT * FROM Movies WHERE year = y AND
        studioName = s)
4) THEN RETURN TRUE;
5) ELSEIF 1 <=
6)     (SELECT COUNT(*) FROM Movies WHERE year = y AND
        studioName = s AND genre = 'comedy')
7) THEN RETURN TRUE;
8) ELSE RETURN FALSE;
9) END IF;

```

Figure 9.13: If there are any movies at all, then at least one has to be a comedy

If studio  $s$  made movies in year  $y$ , then lines (5) and (6) test if at least one of them was a comedy. If so, the return-value is again set to true, this time at line (7). In the remaining case, studio  $s$  made movies but only in color, so we set the return-value to FALSE at line (8).  $\square$

#### 9.4.4 Queries in PSM

There are several ways that select-from-where queries are used in PSM.

1. Subqueries can be used in conditions, or in general, any place a subquery is legal in SQL. We saw two examples of subqueries in lines (3) and (6) of Fig. 9.13, for instance.
2. Queries that return a single value can be used as the right sides of assignment statements.
3. A single-row select statement is a legal statement in PSM. Recall this statement has an INTO clause that specifies variables into which the com-

ponents of the single returned tuple are placed. These variables could be local variables or parameters of a PSM procedure. The general form was discussed in the context of embedded SQL in Section 9.3.5.

4. We can declare and use a cursor, essentially as it was described in Section 9.3.6 for embedded SQL. The declaration of the cursor, OPEN, FETCH, and CLOSE statements are all as described there, with the exceptions that:
  - (a) No EXEC SQL appears in the statements, and
  - (b) The variables do not use a colon prefix.

```
CREATE PROCEDURE SomeProc(IN studioName CHAR(15))

DECLARE presNetWorth INTEGER;

SELECT netWorth
INTO presNetWorth
FROM Studio, MovieExec
WHERE presC# = cert# AND Studio.name = studioName;
...

```

Figure 9.14: A single-row select in PSM

**Example 9.13:** In Fig. 9.14 is the single-row select of Fig. 9.7, redone for PSM and placed in the context of a hypothetical procedure definition. Note that, because the single-row select returns a one-component tuple, we could also get the same effect from an assignment statement, as:

```
SET presNetWorth = (SELECT netWorth
                     FROM Studio, MovieExec
                     WHERE presC# = cert# AND Studio.name = studioName);
```

We shall defer examples of cursor use until we learn the PSM loop statements in the next section. □

#### 9.4.5 Loops in PSM

The basic loop construct in PSM is:

```
LOOP
  <statement list>
END LOOP;
```

One often labels the LOOP statement, so it is possible to break out of the loop, using a statement:

```
LEAVE <loop label>;
```

In the common case that the loop involves the fetching of tuples via a cursor, we often wish to leave the loop when there are no more tuples. It is useful to declare a *condition* name for the SQLSTATE value that indicates no tuple found ('02000', recall); we do so with:

```
DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
```

More generally, we can declare a condition with any desired name corresponding to any SQLSTATE value by

```
DECLARE <name> CONDITION FOR SQLSTATE <value>;
```

We are now ready to take up an example that ties together cursor operations and loops in PSM.

**Example 9.14:** Figure 9.15 shows a PSM procedure that takes a studio name *s* as an input argument and produces in output arguments **mean** and **variance** the mean and variance of the lengths of all the movies owned by studio *s*. Lines (1) through (4) declare the procedure and its parameters.

Lines (5) through (8) are local declarations. We define **Not\_Found** to be the name of the condition that means a **FETCH** failed to return a tuple at line (5). Then, at line (6), the cursor **MovieCursor** is defined to return the set of the lengths of the movies by studio *s*. Lines (7) and (8) declare two local variables that we'll need. Integer **newLength** holds the result of a **FETCH**, while **movieCount** counts the number of movies by studio *s*. We need **movieCount** so that, at the end, we can convert a sum of lengths into an average (**mean**) of lengths and a sum of squares of the lengths into a **variance**.

The rest of the lines are the body of the procedure. We shall use **mean** and **variance** as temporary variables, as well as for “returning” the results at the end. In the major loop, **mean** actually holds the sum of the lengths, and **variance** actually holds the sum of the squares of the lengths. Thus, lines (9) through (11) initialize these variables and the count of the movies to 0. Line (12) opens the cursor, and lines (13) through (19) form the loop labeled **movieLoop**.

Line (14) performs a fetch, and at line (15) we check that another tuple was found. If not, we leave the loop. Lines (16) through (18) accumulate values; we add 1 to **movieCount**, add the length to **mean** (which, recall, is really computing the sum of lengths), and we add the square of the length to **variance**.

When all movies by studio *s* have been seen, we leave the loop, and control passes to line (20). At that line, we turn **mean** into its correct value by dividing the sum of lengths by the count of movies. At line (21), we make **variance** truly hold the variance by dividing the sum of squares of the lengths by the number of movies and subtracting the square of the mean. See Exercise 9.4.4 for a discussion of why this calculation is correct. Line (22) closes the cursor, and we are done. □

```

1) CREATE PROCEDURE MeanVar(
2)     IN s CHAR(15),
3)     OUT mean REAL,
4)     OUT variance REAL
    )
5) DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
6) DECLARE MovieCursor CURSOR FOR
        SELECT length FROM Movies WHERE studioName = s;
7) DECLARE newLength INTEGER;
8) DECLARE movieCount INTEGER;

BEGIN
9)     SET mean = 0.0;
10)    SET variance = 0.0;
11)    SET movieCount = 0;
12)    OPEN MovieCursor;
13)    movieLoop: LOOP
14)        FETCH FROM MovieCursor INTO newLength;
15)        IF Not_Found THEN LEAVE movieLoop END IF;
16)        SET movieCount = movieCount + 1;
17)        SET mean = mean + newLength;
18)        SET variance = variance + newLength * newLength;
19)    END LOOP;
20)    SET mean = mean/movieCount;
21)    SET variance = variance/movieCount - mean * mean;
22)    CLOSE MovieCursor;
END;

```

Figure 9.15: Computing the mean and variance of lengths of movies by one studio

#### 9.4.6 For-Loops

There is also in PSM a for-loop construct, but it is used only to iterate over a cursor. The form of the statement is shown in Fig. 9.16. This statement not only declares a cursor, but it handles for us a number of “grubby details”: the opening and closing of the cursor, the fetching, and the checking whether there are no more tuples to be fetched. However, since we are not fetching tuples for ourselves, we can not specify the variable(s) into which component(s) of a tuple are placed. Thus, the names used for the attributes in the result of the query are also treated by PSM as local variables of the same type.

**Example 9.15:** Let us redo the procedure of Fig. 9.15 using a for-loop. The code is shown in Fig. 9.17. Many things have not changed. The declaration of the procedure in lines (1) through (4) of Fig. 9.17 are the same, as is the

## Other Loop Constructs

PSM also allows while- and repeat-loops, which have the expected meaning, as in C. That is, we can create a loop of the form

```
WHILE <condition> DO
    <statement list>
END WHILE;
```

or a loop of the form

```
REPEAT
    <statement list>
UNTIL <condition>
END REPEAT;
```

Incidentally, if we label these loops, or the loop formed by a loop-statement or for-statement, then we can place the label as well after the END LOOP or other ender. The advantage of doing so is that it makes clearer where each loop ends, and it allows the PSM compiler to catch some syntactic errors involving the omission of an END.

```
FOR <loop name> AS <cursor name> CURSOR FOR
    <query>
DO
    <statement list>
END FOR;
```

Figure 9.16: The PSM for-statement

declaration of local variable `movieCount` at line (5).

However, we no longer need to declare a cursor in the declaration portion of the procedure, and we do not need to define the condition `NotFound`. Lines (6) through (8) initialize the variables, as before. Then, in line (9) we see the for-loop, which also defines the cursor `MovieCursor`. Lines (11) through (13) are the body of the loop. Notice that in lines (12) and (13), we refer to the length retrieved via the cursor by the attribute name `length`, rather than by the local variable name `newLength`, which does not exist in this version of the procedure. Lines (15) and (16) compute the correct values for the output variables, exactly as in the earlier version of this procedure. □

```

1) CREATE PROCEDURE MeanVar(
2)     IN s CHAR(15),
3)     OUT mean REAL,
4)     OUT variance REAL
)
5) DECLARE movieCount INTEGER;

BEGIN
6)     SET mean = 0.0;
7)     SET variance = 0.0;
8)     SET movieCount = 0;
9)     FOR movieLoop AS MovieCursor CURSOR FOR
        SELECT length FROM Movies WHERE studioName = s;
10)    DO
11)        SET movieCount = movieCount + 1;
12)        SET mean = mean + length;
13)        SET variance = variance + length * length;
14)    END FOR;
15)    SET mean = mean/movieCount;
16)    SET variance = variance/movieCount - mean * mean;
END;

```

Figure 9.17: Computing the mean and variance of lengths using a for-loop

#### 9.4.7 Exceptions in PSM

A SQL system indicates error conditions by setting a nonzero sequence of digits in the five-character string `SQLSTATE`. We have seen one example of these codes: '`02000`' for "no tuple found." For another example, '`21000`' indicates that a single-row select has returned more than one row.

PSM allows us to declare a piece of code, called an *exception handler*, that is invoked whenever one of a list of these error codes appears in `SQLSTATE` during the execution of a statement or list of statements. Each exception handler is associated with a block of code, delineated by `BEGIN...END`. The handler appears within this block, and it applies only to statements within the block.

The components of the handler are:

1. A list of exception conditions that invoke the handler when raised.
2. Code to be executed when one of the associated exceptions is raised.
3. An indication of where to go after the handler has finished its work.

The form of a handler declaration is:

```
DECLARE <where to go next> HANDLER FOR <condition list>
    <statement>
```

## Why Do We Need Names in For-Loops?

Notice that `movieLoop` and `MovieCursor`, although declared at line (9) of Fig. 9.17, are never used in that procedure. Nonetheless, we have to invent names, both for the for-loop itself and for the cursor over which it iterates. The reason is that the PSM interpreter will translate the for-loop into a conventional loop, much like the code of Fig. 9.15, and in this code, there is a need for both names.

The choices for “where to go” are:

- a) `CONTINUE`, which means that after executing the statement in the handler declaration, we execute the statement after the one that raised the exception.
- b) `EXIT`, which means that after executing the handler’s statement, control leaves the `BEGIN...END` block in which the handler is declared. The statement after this block is executed next.
- c) `UNDO`, which is the same as `EXIT`, except that any changes to the database or local variables that were made by the statements of the block executed so far are undone. That is, the block is a transaction, which is aborted by the exception.

The “condition list” is a comma-separated list of conditions, which are either declared conditions, like `Not_Found` in line (5) of Fig. 9.15, or expressions of the form `SQLSTATE` and a five-character string.

**Example 9.16:** Let us write a PSM function that takes a movie title as argument and returns the year of the movie. If there is no movie of that title or more than one movie of that title, then `NULL` must be returned. The code is shown in Fig. 9.18.

Lines (2) and (3) declare symbolic conditions; we do not have to make these definitions, and could as well have used the SQL states for which they stand in line (4). Lines (4), (5), and (6) are a block, in which we first declare a handler for the two conditions in which either zero tuples are returned, or more than one tuple is returned. The action of the handler, on line (5), is simply to set the return-value to `NULL`.

Line (6) is the statement that does the work of the function `GetYear`. It is a `SELECT` statement that is expected to return exactly one integer, since that is what the function `GetYear` returns. If there is exactly one movie with title  $t$  (the input parameter of the function), then this value will be returned. However, if an exception is raised at line (6), either because there is no movie with title  $t$  or several movies with that title, then the handler is invoked, and `NULL` instead

```

1) CREATE FUNCTION GetYear(t VARCHAR(255)) RETURNS INTEGER
2) DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
3) DECLARE Too_Many CONDITION FOR SQLSTATE '21000';

BEGIN
4)     DECLARE EXIT HANDLER FOR Not_Found, Too_Many
5)         RETURN NULL;
6)     RETURN (SELECT year FROM Movies WHERE title = t);
END;

```

Figure 9.18: Handling exceptions in which a single-row select returns other than one tuple

becomes the return-value. Also, since the handler is an EXIT handler, control next passes to the point after the END. Since that point is the end of the function, `GetYear` returns at that time, with the return-value NULL. □

#### 9.4.8 Using PSM Functions and Procedures

As we mentioned in Section 9.4.2, we can call a PSM procedure anywhere SQL statements can appear, e.g., as embedded SQL, from PSM code itself, or from SQL issued to the generic interface. We invoke a procedure by preceding it by the keyword CALL. In addition, a PSM function can be used as part of an expression, e.g., in a WHERE clause. Here is an example of how a function can be used within an expression.

**Example 9.17:** Suppose that our schema includes a module with the function `GetYear` of Fig. 9.18. Imagine that we are sitting at the generic interface, and we want to enter the fact that Denzel Washington was a star of *Remember the Titans*. However, we forget the year in which that movie was made. As long as there was only one movie of that name, and it is in the `Movies` relation, we don't have to look it up in a preliminary query. Rather, we can issue to the generic SQL interface the following insertion:

```

INSERT INTO StarsIn(movieTitle, movieYear, starName)
VALUES('Remember the Titans', GetYear('Remember the Titans'),
      'Denzel Washington');

```

Since `GetYear` returns NULL if there is not a unique movie by the name of *Remember the Titans*, it is possible that this insertion will have NULL in the middle component. □

#### 9.4.9 Exercises for Section 9.4

**Exercise 9.4.1:** Using our running movie database:

```

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

```

write PSM procedures or functions to perform the following tasks:

- a) Given the name of a movie studio, produce the net worth of its president.
- b) Given a name and address, return 1 if the person is a movie star but not an executive, 2 if the person is an executive but not a star, 3 if both, and 4 if neither.
- c) Given a studio name, assign to output parameters the titles of the two longest movies by that studio. Assign NULL to one or both parameters if there is no such movie (e.g., if there is only one movie by a studio, there is no “second-longest”).
- d) Given a star name, find the earliest (lowest year) movie of more than 120 minutes length in which they appeared. If there is no such movie, return the year 0.
- e) Given an address, find the name of the unique star with that address if there is exactly one, and return NULL if there is none or more than one.
- f) Given the name of a star, delete them from MovieStar and delete all their movies from StarsIn and Movies.

**Exercise 9.4.2:** Write the following PSM functions or procedures, based on the database schema

```

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

```

of Exercise 2.4.1.

- a) Take a price as argument and return the model number of the PC whose price is closest.
- b) Take a maker and model as arguments, and return the price of whatever type of product that model is.
- c) Take model, speed, ram, hard-disk, and price information as arguments, and insert this information into the relation PC. However, if there is already a PC with that model number (tell by assuming that violation of a key constraint on insertion will raise an exception with SQLSTATE equal to '23000'), then keep adding 1 to the model number until you find a model number that is not already a PC model number.

- ! d) Given a price, produce the number of PC's, the number of laptops, and the number of printers selling for more than that price.

**Exercise 9.4.3:** Write the following PSM functions or procedures, based on the database schema

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

of Exercise 2.4.3.

- a) The firepower of a ship is roughly proportional to the number of guns times the cube of the bore. Given a class, find its firepower.
  - ! b) Given the name of a battle, produce the two countries whose ships were involved in the battle. If there are more or fewer than two countries involved, produce NULL for both countries.
  - c) Take as arguments a new class name, type, country, number of guns, bore, and displacement. Add this information to `Classes` and also add the ship with the class name to `Ships`.
  - ! d) Given a ship name, determine if the ship was in a battle with a date before the ship was launched. If so, set the date of the battle and the date the ship was launched to 0.
- ! **Exercise 9.4.4:** In Fig. 9.15, we used a tricky formula for computing the variance of a sequence of numbers  $x_1, x_2, \dots, x_n$ . Recall that the variance is the average square of the deviation of these numbers from their mean. That is, the variance is  $(\sum_{i=1}^n (x_i - \bar{x})^2)/n$ , where the mean  $\bar{x}$  is  $(\sum_{i=1}^n x_i)/n$ . Prove that the formula for the variance used in Fig. 9.15, which is

$$\left( \sum_{i=1}^n (x_i)^2 \right)/n - \left( \left( \sum_{i=1}^n x_i \right)/n \right)^2$$

yields the same value.

## 9.5 Using a Call-Level Interface

When using a *call-level interface* (CLI), we write ordinary host-language code, and we use a library of functions that allow us to connect to and access a database, passing SQL statements to that database. The differences between this approach and embedded SQL programming are, in one sense, cosmetic, since the preprocessor replaces embedded SQL by calls to library functions much like the functions in the standard SQL/CLI.

We shall give three examples of call-level interfaces. In this section, we cover the standard SQL/CLI, which is an adaptation of ODBC (Open Database Connectivity). We cover JDBC, which is a collection of classes that support database access from Java programs. Then, we explore PHP, which is a way to embed database access in Web pages described by HTML.

### 9.5.1 Introduction to SQL/CLI

A program written in C and using SQL/CLI (hereafter, just CLI) will include the header file `sqlcli.h`, from which it gets a large number of functions, type definitions, structures, and symbolic constants. The program is then able to create and deal with four kinds of records (structs, in C):

1. *Environments*. A record of this type is created by the application (client) program in preparation for one or more connections to the database server.
2. *Connections*. One of these records is created to connect the application program to the database. Each connection exists within some environment.
3. *Statements*. An application program can create one or more statement records. Each holds information about a single SQL statement, including an implied cursor if the statement is a query. At different times, the same CLI statement can represent different SQL statements. Every CLI statement exists within some connection.
4. *Descriptions*. These records hold information about either tuples or parameters. The application program or the database server, as appropriate, sets components of description records to indicate the names and types of attributes and/or their values. Each statement has several of these created implicitly, and the user can create more if needed. In our presentation of CLI, description records will generally be invisible.

Each of these records is represented in the application program by a *handle*, which is a pointer to the record. The header file `sqlcli.h` provides types for the handles of environments, connections, statements, and descriptions: `SQLHENV`, `SQLHDBC`, `SQLHSTMT`, and `SQLHDESC`, respectively, although we may think of them as pointers or integers. We shall use these types and also some other defined types with obvious interpretations, such as `SQL_CHAR` and `SQL_INTEGER`, that are provided in `sqlcli.h`.

We shall not go into detail about how descriptions are set and used. However, (handles for) the other three types of records are created by the use of a function

```
SQLAllocHandle(hType, hIn, hOut)
```

Here, the three arguments are:

1. *hType* is the type of handle desired. Use `SQL_HANDLE_ENV` for a new environment, `SQL_HANDLE_DBC` for a new connection, or `SQL_HANDLE_STMT` for a new statement.
2. *hIn* is the handle of the higher-level element in which the newly allocated element lives. This parameter is `SQL_NULL_HANDLE` if you want an environment; the latter name is a defined constant telling `SQLAllocHandle` that there is no relevant value here. If you want a connection handle, then *hIn* is the handle of the environment within which the connection will exist, and if you want a statement handle, then *hIn* is the handle of the connection within which the statement will exist.
3. *hOut* is the address of the handle that is created by `SQLAllocHandle`.

`SQLAllocHandle` also returns a value of type `SQLRETURN` (an integer). This value is 0 if no errors occurred, and there are certain nonzero values returned in the case of errors.

**Example 9.18:** Let us see how the function `worthRanges` of Fig. 9.8, which we used as an example of embedded SQL, would begin in CLI. Recall this function examines all the tuples of `MovieExec` and breaks their net worths into ranges. The initial steps are shown in Fig. 9.19.

```

1) #include sqlcli.h
2) SQLHENV myEnv;
3) SQLHDBC myCon;
4) SQLHSTMT execStat;
5) SQLRETURN errorCode1, errorCode2, errorCode3;

6) errorCode1 = SQLAllocHandle(SQL_HANDLE_ENV,
                               SQL_NULL_HANDLE, &myEnv);
7) if(!errorCode1) {
8)     errorCode2 = SQLAllocHandle(SQL_HANDLE_DBC,
                               myEnv, &myCon);
9)     if(!errorCode2)
10)        errorCode3 = SQLAllocHandle(SQL_HANDLE_STMT,
                               myCon, &execStat); }
```

5

Figure 9.19: Declaring and creating an environment, a connection, and a statement

Lines (2) through (4) declare handles for an environment, connection, and statement, respectively; their names are `myEnv`, `myCon`, and `execStat`, respectively. We plan that `execStat` will represent the SQL statement

```
SELECT netWorth FROM MovieExec;
```

much as did the cursor `execCursor` in Fig. 9.8, but as yet there is no SQL statement associated with `execStat`. Line (5) declares three variables into which function calls can place their response and indicate an error. A value of 0 indicates no error occurred in the call.

Line (6) calls `SQLAllocHandle`, asking for an environment handle (the first argument), providing a null handle in the second argument (because none is needed when we are requesting an environment handle), and providing the address of `myEnv` as the third argument; the generated handle will be placed there. If line (6) is successful, lines (7) and (8) use the environment handle to get a connection handle in `myCon`. Assuming that call is also successful, lines (9) and (10) get a statement handle for `execStat`. □

## 9.5.2 Processing Statements

At the end of Fig. 9.19, a statement record whose handle is `execStat`, has been created. However, there is as yet no SQL statement with which that record is associated. The process of associating and executing SQL statements with statement handles is analogous to the dynamic SQL described in Section 9.3.9. There, we associated the text of a SQL statement with what we called a “SQL variable,” using `PREPARE`, and then executed it using `EXECUTE`.

The situation in CLI is quite analogous, if we think of the “SQL variable” as a statement handle. There is a function

`SQLPrepare(sh, st, sl)`

that takes:

1. A statement handle `sh`,
2. A pointer to a SQL statement `st`, and
3. A length `sl` for the character string pointed to by `st`. If we don’t know the length, a defined constant `SQL_NTS` tells `SQLPrepare` to figure it out from the string itself. Presumably, the string is a “null-terminated string,” and it is sufficient for `SQLPrepare` to scan it until encountering the endmarker ‘\0’.

The effect of this function is to arrange that the statement referred to by the handle `sh` now represents the particular SQL statement `st`.

Another function

`SQLExecute(sh)`

causes the statement to which handle `sh` refers to be executed. For many forms of SQL statement, such as insertions or deletions, the effect of executing this statement on the database is obvious. Less obvious is what happens when the SQL statement referred to by `sh` is a query. As we shall see in Section 9.5.3,

there is an implicit cursor for this statement that is part of the statement record itself. The statement is in principle executed, so we can imagine that all the answer tuples are sitting somewhere, ready to be accessed. We can fetch tuples one at a time, using the implicit cursor, much as we did with real cursors in Sections 9.3 and 9.4.

**Example 9.19:** Let us continue with the function `worthRanges` that we began in Fig. 9.19. The following two function calls associate the query

```
SELECT netWorth FROM MovieExec;
```

with the statement referred to by handle `execStat`:

- 11) `SQLPrepare(execStat, "SELECT netWorth FROM MovieExec", SQL_NTS);`
- 12) `SQLExecute(execStat);`

These lines could appear right after line (10) of Fig. 9.19. Remember that `SQL_NTS` tells `SQLPrepare` to determine the length of the null-terminated string to which its second argument refers. □

As with dynamic SQL, the prepare and execute steps can be combined into one if we use the function `SQLExecDirect`. An example that combines lines (11) and (12) above is:

```
SQLExecDirect(execStat, "SELECT netWorth FROM MovieExec",
    SQL_NTS);
```

### 9.5.3 Fetching Data From a Query Result

The function that corresponds to a `FETCH` command in embedded SQL or PSM is

`SQLFetch(sh)`

where `sh` is a statement handle. We presume the statement referred to by `sh` has been executed already, or the fetch will cause an error. `SQLFetch`, like all CLI functions, returns a value of type `SQLRETURN` that indicates either success or an error. The return value `SQL_NO_DATA` tells us tuples were left in the query result. As in our previous examples of fetching, this value will be used to get us out of a loop in which we repeatedly fetch new tuples from the result.

However, if we follow the `SQLExecute` of Example 9.19 by one or more `SQLFetch` calls, where does the tuple appear? The answer is that its components go into one of the description records associated with the statement whose handle appears in the `SQLFetch` call. We can extract the same component at each fetch by binding the component to a host-language variable, before we begin fetching. The function that does this job is:

`SQLBindCol(sh, colNo, colType, pVar, varSize, varInfo)`

The meanings of these six arguments are:

1. *sh* is the handle of the statement involved.
2. *colNo* is the number of the component (within the tuple) whose value we obtain.
3. *colType* is a code for the type of the variable into which the value of the component is to be placed. Examples of codes provided by `sqlcli.h` are `SQL_CHAR` for character arrays and strings, and `SQL_INTEGER` for integers.
4. *pVar* is a pointer to the variable into which the value is to be placed.
5. *varSize* is the length in bytes of the value of the variable pointed to by *pVar*.
6. *varInfo* is a pointer to an integer that can be used by `SQLBindCol` to provide additional information about the value produced.

**Example 9.20:** Let us redo the entire function `worthRanges` from Fig. 9.8, using CLI calls instead of embedded SQL. We begin as in Fig. 9.19, but for the sake of succinctness, we skip all error checking except for the test whether `SQLFetch` indicates that no more tuples are present. The code is shown in Fig. 9.20.

Line (3) declares the same local variables that the embedded-SQL version of the function uses, and lines (4) through (7) declare additional local variables using the types provided in `sqlcli.h`; these are variables that involve SQL in some way. Lines (4) through (6) are as in Fig. 9.19. New are the declarations on line (7) of `worth` (which corresponds to the shared variable of that name in Fig. 9.8) and `worthInfo`, which is required by `SQLBindCol`, but not used.

Lines (8) through (10) allocate the needed handles, as in Fig. 9.19, and lines (11) and (12) prepare and execute the SQL statement, as discussed in Example 9.19. In line (13), we see the binding of the first (and only) column of the result of this query to the variable `worth`. The first argument is the handle for the statement involved, and the second argument is the column involved, 1 in this case. The third argument is the type of the column, and the fourth argument is a pointer to the place where the value will be placed: the variable `worth`. The fifth argument is the size of that variable, and the final argument points to `worthInfo`, a place for `SQLBindCol` to put additional information (which we do not use here).

The balance of the function resembles closely lines (11) through (19) of Fig. 9.8. The while-loop begins at line (14) of Fig. 9.20. Notice that we fetch a tuple and check that we are not out of tuples, all within the condition of the while-loop, on line (14). If there is a tuple, then in lines (15) through (17) we determine the number of digits the integer (which is bound to `worth`) has and increment the appropriate count. After the loop finishes, i.e., all tuples returned

```

1) #include sqlcli.h
2) void worthRanges() {

3)     int i, digits, counts[15];
4)     SQLHENV myEnv;
5)     SQLHDBC myCon;
6)     SQLHSTMT execStat;
7)     SQLINTEGER worth, worthInfo;

8)     SQLAllocHandle(SQL_HANDLE_ENV,
9)         SQL_NULL_HANDLE, &myEnv);
10)    SQLAllocHandle(SQL_HANDLE_DBC, myEnv, &myCon);
11)    SQLAllocHandle(SQL_HANDLE_STMT, myCon, &execStat);
12)    SQLPrepare(execStat,
13)        "SELECT netWorth FROM MovieExec", SQL_NTS);
14)    SQLExecute(execStat);
15)    SQLBindCol(execStat, 1, SQL_INTEGER, &worth,
16)        sizeof(worth), &worthInfo);
17)    while(SQLFetch(execStat) != SQL_NO_DATA) {
18)        digits = 1;
19)        while((worth /= 10) > 0) digits++;
20)        if(digits <= 14) counts[digits]++;
21)
22)    for(i=0; i<15; i++)
23)        printf("digits = %d: number of execs = %d\n",
24)            i, counts[i]);
25}

```

Figure 9.20: Grouping executive net worths: CLI version

by the statement execution of line (12) have been examined, the resulting counts are printed out at lines (18) and (19). □

#### 9.5.4 Passing Parameters to Queries

Embedded SQL gives us the ability to execute a SQL statement, part of which consists of values determined by the current contents of shared variables. There is a similar capability in CLI, but it is rather more complicated. The steps needed are:

1. Use `SQLPrepare` to prepare a statement in which some portions, called *parameters*, are replaced by a question-mark. The *i*th question-mark represents the *i*th parameter.

## Extracting Components with SQLGetData

An alternative to binding a program variable to an output of a query's result relation is to fetch tuples without any binding and then transfer components to program variables as needed. The function to use is **SQLGetData**, and it takes the same arguments as **SQLBindCol**. However, it only copies data once, and it must be used after each fetch in order to have the same effect as initially binding the column to a variable.

2. Use function **SQLBindParameter** to bind values to the places where the question-marks are found. This function has ten arguments, of which we shall explain only the essentials.
3. Execute the query with these bindings, by calling **SQLExecute**. Note that if we change the values of one or more parameters, we need to call **SQLExecute** again.

The following example will illustrate the process, as well as indicate the important arguments needed by **SQLBindParameter**.

**Example 9.21:** Let us reconsider the embedded SQL code of Fig. 9.6, where we obtained values for two variables **studioName** and **studioAddr** and used them as the components of a tuple, which we inserted into **Studio**. Figure 9.21 sketches how this process would work in CLI. It assumes that we have a statement handle **myStat** to use for the insertion statement.

```

/* get values for studioName and studioAddr */

1) SQLPrepare(myStat,
               "INSERT INTO Studio(name, address) VALUES(?, ?)",
               SQL_NTS);
2) SQLBindParameter(myStat, 1, ..., studioName,...);
3) SQLBindParameter(myStat, 2, ..., studioAddr,...);
4) SQLExecute(myStat);

```

Figure 9.21: Inserting a new studio by binding parameters to values

The code begins with steps (not shown) to give **studioName** and **studioAddr** values. Line (1) shows statement **myStat** being prepared to be an insertion statement with two parameters (the question-marks) in the **VALUE** clause. Then, lines (2) and (3) bind the first and second question-marks, to the current contents of **studioName** and **studioAddr**, respectively. Finally, line (4) executes the insertion. If the entire sequence of steps in Fig. 9.21, including the unseen work to obtain new values for **studioName** and **studioAddr**, are placed

in a loop, then each time around the loop, a new tuple, with a new name and address for a studio, is inserted into `Studio`. □

### 9.5.5 Exercises for Section 9.5

**Exercise 9.5.1:** Repeat the problems of Exercise 9.3.1, but write the code in C with CLI calls.

**Exercise 9.5.2:** Repeat the problems of Exercise 9.3.2, but write the code in C with CLI calls.

## 9.6 JDBC

*Java Database Connectivity*, or JDBC, is a facility similar to CLI for allowing Java programs to access SQL databases. The concepts resemble those of CLI, although Java's object-oriented flavor is evident in JDBC.

### 9.6.1 Introduction to JDBC

The first steps we must take to use JDBC are:

1. include the line:

```
import java.sql.*;
```

to make the JDBC classes available to your Java program.

2. Load a “driver” for the database system we shall use. The driver we need depends on which DBMS is available to us, but we load the needed driver with the statement:

```
Class.forName(<driver name>);
```

For example, to get the driver for a MySQL database, execute:

```
Class.forName("com.mysql.jdbc.Driver");
```

The effect is that a class called `DriverManager` is available. This class is analogous in many ways to the environment whose handle we get as the first step in using CLI.

3. Establish a connection to the database. A variable of class `Connection` is created if we apply the method `getConnection` to `DriverManager`.

The Java statement to establish a connection looks like:

```
Connection myCon = DriverManager.getConnection(<URL>,
                                             <user name>, <password>);
```

That is, the method `getConnection` takes as arguments the URL for the database to which you wish to connect, your user name, and your password. It returns an object of class `Connection`, which we have chosen to call `myCon`.

**Example 9.22:** Each DBMS has its own way of specifying the URL in the `getConnection` method. For instance, if you want to connect to a MySQL database, the form of the URL is

```
. jdbc:mysql://<host name>/<database name>
```

□

A JDBC Connection object is quite analogous to a CLI connection, and it serves the same purpose. By applying the appropriate methods to a `Connection` like `myCon`, we can create statement objects, place SQL statements “in” those objects, bind values to SQL statement parameters, execute the SQL statements, and examine results a tuple at a time.

## 9.6.2 Creating Statements in JDBC

There are two methods we can apply to a `Connection` object in order to create statements:

1. `createStatement()` returns a `Statement` object. This object has no associated SQL statement yet, so method `createStatement()` may be thought of as analogous to the CLI call to `SQLAllocHandle` that takes a connection handle and returns a statement handle.
2. `prepareStatement(Q)`, where *Q* is a SQL query passed as a string argument, returns a `PreparedStatement` object. Thus, we may draw an analogy between executing `prepareStatement(Q)` in JDBC with the two CLI steps in which we get a statement handle with `SQLAllocHandle` and then apply `SQLPrepare` to that handle and the query *Q*.

There are four different methods that execute SQL statements. Like the methods above, they differ in whether or not they take a SQL statement as an argument. However, these methods also distinguish between SQL statements that are queries and other statements, which are collectively called “updates.” Note that the SQL UPDATE statement is only one small example of what JDBC terms an “update.” The latter include all modification statements, such as inserts, and all schema-related statements such as CREATE TABLE. The four “execute” methods are:

- a) `executeQuery(Q)` takes a statement *Q*, which must be a query, and is applied to a Statement object. This method returns a ResultSet object, which is the set (bag, to be precise) of tuples produced by the query *Q*. We shall see how to access these tuples in Section 9.6.3.
- b) `executeQuery()` is applied to a PreparedStatement object. Since a prepared statement already has an associated query, there is no argument. This method also returns a ResultSet object.
- c) `executeUpdate(U)` takes a nonquery statement *U* and, when applied to a Statement object, executes *U*. The effect is felt on the database only; no ResultSet object is returned.
- d) `executeUpdate()`, with no argument, is applied to a PreparedStatement object. In that case, the SQL statement associated with the prepared statement is executed. This SQL statement must not be a query, of course.

**Example 9.23:** Suppose we have a Connection object `myCon`, and we wish to execute the query

```
SELECT netWorth FROM MovieExec;
```

One way to do so is to create a Statement object `execStat`, and then use it to execute the query directly.

```
Statement execStat = myCon.createStatement();
ResultSet worths = execStat.executeQuery(
    "SELECT netWorth FROM MovieExec");
```

The result of the query is a ResultSet object, which we have named `worths`. We'll see in Section 9.6.3 how to extract the tuples from `worths` and process them.

An alternative is to prepare the query immediately and later execute it. This approach would be preferable should we want to execute the same query repeatedly. Then, it makes sense to prepare it once and execute it many times, rather than having the DBMS prepare the same query many times. The JDBC steps needed to follow this approach are:

```
PreparedStatement execStat = myCon.prepareStatement(
    "SELECT netWorth FROM MovieExec");
ResultSet worths = execStat.executeQuery();
```

The result of executing the query is again a ResultSet object, which we have called `worths`.  $\square$

**Example 9.24:** If we want to execute a parameterless nonquery, we can perform analogous steps in both styles. There is no result set, however. For instance, suppose we want to insert into `StarsIn` the fact that Denzel Washington starred in *Remember the Titans* in the year 2000. We may create and use a statement `starStat` in either of the following ways:

```
Statement starStat = myCon.createStatement();
starStat.executeUpdate("INSERT INTO StarsIn VALUES(" +
    "'Remember the Titans', 2000, 'Denzel Washington')");
```

or

```
PreparedStatement starStat = myCon.prepareStatement(
    "INSERT INTO StarsIn VALUES('Remember the Titans', " +
    "2000, 'Denzel Washington')");
starStat.executeUpdate();
```

Notice that each of these sequences of Java statements takes advantage of the fact that `+` is the Java operator that concatenates strings. Thus, we are able to extend SQL statements over several lines of Java, as needed. □

### 9.6.3 Cursor Operations in JDBC

When we execute a query and obtain a result-set object, we may, in effect, run a cursor through the tuples of the result set. To do so, the `ResultSet` class provides the following useful methods:

1. `next()`, when applied to a `ResultSet` object, causes an implicit cursor to move to the next tuple (to the first tuple the first time it is applied). This method returns `FALSE` if there is no next tuple.
2. `getString(i)`, `getInt(i)`, `getFloat(i)`, and analogous methods for the other types that SQL values can take, each return the *i*th component of the tuple currently indicated by the cursor. The method appropriate to the type of the *i*th component must be used.

**Example 9.25:** Having obtained the result set `worths` as in Example 9.23, we may access its tuples one at a time. Recall that these tuples have only one component, of type integer. The form of the loop is:

```
while(worths.next()) {
    int worth = worths.getInt(1);
    /* process this net worth */
};
```

□

## 9.6.4 Parameter Passing

As in CLI, we can use a question-mark in place of a portion of a query, and then bind values to those *parameters*. To do so in JDBC, we need to create a prepared statement, and we need to apply to that PreparedStatement object methods such as `setString(i, v)` or `setInt(i, v)` that bind the value *v*, which must be of the appropriate type for the method, to the *i*th parameter in the query.

**Example 9.26:** Let us mimic the CLI code in Example 9.21, where we prepared a statement to insert a new studio into relation `Studio`, with parameters for the name and address of that studio. The Java code to prepare this statement, set its parameters, and execute it is shown in Fig. 9.22. We continue to assume that connection object `myCon` is available to us.

```

1) PreparedStatement studioStat = myCon.prepareStatement(
2)       "INSERT INTO Studio(name, address) VALUES(?, ?)";
   /* get values for variables studioName and studioAddr
      from the user */
3) studioStat.setString(1, studioName);
4) studioStat.setString(2, studioAddr);
5) studioStat.executeUpdate();

```

Figure 9.22: Setting and using parameters in JDBC

In lines (1) and (2), we create and prepare the insertion statement. It has parameters for each of the values to be inserted. After line (2), we could begin a loop in which we repeatedly ask the user for a studio name and address, and place these strings in the variables `studioName` and `studioAddr`. This assignment is not shown, but represented by a comment. Lines (3) and (4) set the first and second parameters to the strings that are the current values of `studioName` and `studioAddr`, respectively. Finally, at line (5), we execute the insertion statement with the current values of its parameters. After line (5), we could go around the loop again, beginning with the steps represented by the comment. □

## 9.6.5 Exercises for Section 9.6

**Exercise 9.6.1:** Repeat Exercise 9.3.1, but write the code in Java using JDBC.

**Exercise 9.6.2:** Repeat Exercise 9.3.2, but write the code in Java using JDBC.

## 9.7 PHP

PHP is a scripting language for helping to create HTML Web pages. It provides support for database operations through an available library, much as JDBC

## What Does PHP Stand For?

Originally, PHP was an acronym for “Personal Home Page.” More recently, it is said to be the recursive acronym “PHP: Hypertext Preprocessor” in the spirit of other recursive acronyms such as GNU (= “GNU is Not Unix”).

does. In this section we shall give a brief overview of PHP and show how database operations are performed in this language.

### 9.7.1 PHP Basics

All PHP code is intended to exist inside HTML text. A browser will recognize that text is PHP code by placing it inside a special tag, which looks like:

```
<?php  
    PHP code goes here  
?>
```

Many aspects of PHP, such as assignment statements, branches, and loops, will be familiar to the C or Java programmer, and we shall not cover them explicitly. However, there are some interesting features of PHP of which we should be aware.

### Variables

Variables are untyped and need not be declared. All variable names begin with \$.

Often, a variable will be declared to be a member of a “class,” in which case certain *functions* (analogous to methods in Java) may be applied to that variable. The function-application operator is ->, comparable to the dot in Java or C++.

### Strings

String values in PHP can be surrounded by either single or double quotes, but there is an important difference. Strings surrounded by single quotes are treated literally, just like SQL strings. However, when a string has double quotes around it, any variable names within the string are replaced by their values.

**Example 9.27:** In the following code:

```
$foo = 'bar';  
$x = 'Step up to the $foo';
```

the value of `$x` is Step up to the `$foo`. However, if the following code is executed instead:

```
$foo = "bar";
$x = "Step up to the $foo";
```

the value of `$x` is Step up to the bar. It doesn't matter whether bar has single or double quotes, since it contains no dollar-signs and therefore no variables. However, the variable `$foo` is replaced only when surrounded by double quotes, as in the second example. □

Concatenation of strings is denoted by a dot. Thus,

```
$y = "$foo" . 'bar';
```

gives `$y` the value barbar.

### 9.7.2 Arrays

PHP has ordinary arrays (called *numeric*), which are indexed 0, 1, . . . . It also has arrays that are really mappings, called *associative arrays*. The indexes (*keys*) of an associative array can be any strings, and the array associates a single value with each key. Both kinds of arrays use the conventional square brackets for indexing, but for associative arrays, an array element is represented by:

$$<\text{key}> \Rightarrow <\text{value}>$$

**Example 9.28:** The following line:

```
$a = array(30,20,10,0);
```

sets `$a` to be a numeric array of length four, with `$a[0]` equal to 30, `$a[1]` equal to 20, and so on. □

**Example 9.29:** The following line:

```
$seasons = array('spring' => 'warm', 'summer' => 'hot',
                 'fall' => 'warm', 'winter' => 'cold');
```

makes `$seasons` be an array of length four, but it is an associative array. For instance, `$seasons['summer']` has the value 'hot'. □

### 9.7.3 The PEAR DB Library

PHP has a collection of libraries called PEAR (PHP Extension and Application Repository). One of these libraries, DB, has generic functions that are analogous to the methods of JDBC. We tell the function `DB::connect` which vendor's DBMS we wish to access, but none of the other functions of DB need to know about which DBMS we are using. Note that the double colon in `DB::connect` is PHP's way of saying "the function `connect` in the DB library." We make the DB library available to our PHP program with the statement:

```
include(DB.php);
```

### 9.7.4 Creating a Database Connection Using DB

The form of an invocation of the `connect` function is:

```
$myCon = DB::connect(<vendor>://<user name>:<password>
                      <host name>/<database name>);
```

The components of this call are like those in the analogous JDBC statement that creates a connection (see Section 9.6.1). The one exception is the vendor, which is a code used by the DB library. For example, `mysqli` is the code for recent versions of the MySQL database.

After executing this statement, the variable `$myCon` is a connection. Like all PHP variables, `$myCon` can change its type. But as long as it is a connection, we may apply to it a number of useful functions that enable us to manipulate the database to which the connection was made. For example, we can disconnect from the database by

```
$myCon->disconnect();
```

Remember that `->` is the PHP way of applying a function to an "object."

### 9.7.5 Executing SQL Statements

All SQL statements are referred to as "queries" and are executed by the function `query`, which takes the statement as an argument and is applied to the connection variable.

**Example 9.30:** Let us duplicate the insertion statement of Example 9.24, where we inserted Denzel Washington and *Remember the Titans* into the Stars-In table. Assuming that `$myCon` has connected to our movie database, We can simply say:

```
$result = $myCon->query("INSERT INTO StarsIn VALUES(' .
                           ''Denzel Washington', 2000, 'Remember the Titans')");
```

Note that the dot concatenates the two strings that form the query. We only broke the query into two strings because it was necessary to break it over two lines.

The variable `$result` will hold an error code if the insert-statement failed to execute. If the “query” were really a SQL query, then `$result` is a cursor to the tuples of the result (see Section 9.7.6). □

PHP allows SQL to have parameters, denoted by question-marks, as we shall discuss in Section 9.7.7. However, the ability to expand variables in doubly quoted strings gives us another easy way to execute SQL statements that depend on user input. In particular, since PHP is used within Web pages, there are built-in ways to exploit HTML’s capabilities.

We often get information from a user of a Web page by showing them a form and having their answers “posted.” PHP provides an associative array called `$_POST` with all the information provided by the user. Its keys are the names of the form elements, and the associated values are what the user has entered into the form.

**Example 9.31:** Suppose we ask the user to fill out a form whose elements are `title`, `year`, and `starName`. These three values will form a tuple that we may insert into the table `StarsIn`. The statement:

```
$result = $myCon->query("INSERT INTO StarsIn VALUES(
    $_POST['title'], $_POST['year'], $_POST['starName'])");
```

will obtain the posted values for these three form elements. Since the query argument is a double-quoted string, PHP evaluates terms like `$_POST['title']` and replaces them by their values. □

## 9.7.6 Cursor Operations in PHP

When the `query` function gets a true query as argument, it returns a result object, that is, a list of tuples. Each tuple is a numeric array, indexed by integers starting at 0. The essential function that we can apply to a result object is `fetchRow()`, which returns the next row, or 0 (false) if there is no next row.

```
1) $worths = $myCon->query("SELECT netWorth FROM MovieExec");
2) while ($tuple = $worths->fetchRow()) {
3)     $worth = $tuple[0];
        // process this value of $worth
}
```

Figure 9.23: Finding and processing net worths in PHP

**Example 9.32:** In Fig. 9.23 is PHP code that is the equivalent of the JDBC in Examples 9.23 and 9.25. It assumes that connection `$myCon` is available, as before.

Line (1) passes the query to the connection `$myCon`, and the result object is assigned to the variable `$worths`. We then enter a loop, in which we repeatedly get a tuple from the result and assign this tuple to the variable `$tuple`, which technically becomes an array of length 1, with only a component for the column `netWorth`. As in C, the value returned by `fetchRow()` becomes the value of the condition in the while-statement. Thus, if no tuple is found, this value, 0, terminates the loop. At line (3), the value of the tuple's first (and only) component is extracted and assigned to the variable `$worth`. We do not show the processing of this value. □

### 9.7.7 Dynamic SQL in PHP

As in JDBC, PHP allows a SQL query to contain question-marks. These question-marks are placeholders for values that can be filled in later, during the execution of the statement. The process of doing so is as follows.

We may apply `prepare` and `execute` functions to a connection; these functions are analogous to similarly named functions discussed in Section 9.3.9 and elsewhere. Function `prepare` takes a SQL statement as argument and returns a prepared version of that statement. Function `execute` takes two arguments: the prepared statement and an array of values to substitute for the question-marks in the statement. If there is only one question-mark, a simple variable, rather than an array, suffices.

**Example 9.33:** Let us again look at the problem of Example 9.26, where we prepared to insert many name-address pairs into relation `Studio`. To begin, we prepare the query, with parameters, by:

```
$prepQuery = $myCon->query("INSERT INTO Studio(name, "
    "address) VALUES(?,?)");
```

Now, `$prepQuery` is a “prepared query.” We can use it as an argument to `execute` along with an array of two values, a studio name and address. For example, we could perform the following statements:

```
$args = array('MGM', 'Los Angeles');
$result = $myCon->execute($prepQuery, $args);
```

The advantage of this arrangement is the same as for all implementations of dynamic SQL. If we insert many different tuples this way, we only have to prepare the insertion statement once and can execute it many times. □

### 9.7.8 Exercises for Section 9.7

**Exercise 9.7.1:** Repeat Exercise 9.3.1, but write the code using PHP.

**Exercise 9.7.2:** Repeat Exercise 9.3.2, but write the code using PHP.

**! Exercise 9.7.3:** In Example 9.31 we exploited the feature of PHP that strings in double-quotes have variables expanded. How essential is this feature? Could we have done something analogous in JDBC? If so, how?

## 9.8 Summary of Chapter 9

- ◆ *Three-Tier Architectures:* Large database installations that support large-scale user interactions over the Web commonly use three tiers of processes: web servers, application servers, and database servers. There can be many processes active at each tier, and these processes can be at one processor or distributed over many processors.
- ◆ *Client-Server Systems in the SQL Standard:* The standard talks of SQL clients connecting to SQL servers, creating a connection (link between the two processes) and a session (sequence of operations). The code executed during the session comes from a module, and the execution of the module is called a SQL agent.
- ◆ *The Database Environment:* An installation using a SQL DBMS creates a SQL environment. Within the environment, database elements such as relations are grouped into (database) schemas, catalogs, and clusters. A catalog is a collection of schemas, and a cluster is the largest collection of elements that one user may see.
- ◆ *Impedance Mismatch:* The data model of SQL is quite different from the data models of conventional host languages. Thus, information passes between SQL and the host language through shared variables that can represent components of tuples in the SQL portion of the program.
- ◆ *Embedded SQL:* Instead of using a generic query interface to express SQL queries and modifications, it is often more effective to write programs that embed SQL queries in a conventional host language. A preprocessor converts the embedded SQL statements into suitable function calls of the host language.
- ◆ *Cursors:* A cursor is a SQL variable that indicates one of the tuples of a relation. Connection between the host language and SQL is facilitated by having the cursor range over each tuple of the relation, while the components of the current tuple are retrieved into shared variables and processed using the host language.

- ◆ **Dynamic SQL:** Instead of embedding particular SQL statements in a host-language program, the host program may create character strings that are interpreted by the SQL system as SQL statements and executed.
- ◆ **Persistent Stored Modules:** We may create collections of procedures and functions as part of a database schema. These are written in a special language that has all the familiar control primitives, as well as SQL statements.
- **The Call-Level Interface:** There is a standard library of functions, called SQL/CLI or ODBC, that can be linked into any C program. These functions give capabilities similar to embedded SQL, but without the need for a preprocessor.
- ◆ **JDBC:** Java Database Connectivity is a collection of Java classes analogous to CLI for connecting Java programs to a database.
- ◆ **PHP:** Another popular system for implementing a call-level interface is PHP. This language is found embedded in HTML pages and enables these pages to interact with a database.

## 9.9 References for Chapter 9

The PSM standard is [4], and [5] is a comprehensive book on the subject. Oracle's version of PSM is called PL/SQL; a summary can be found in [2]. SQL Server has a version called Transact-SQL [6]. IBM's version is SQL PL [1].

[3] is a popular reference on JDBC. [7] is one on PHP, which was originally developed by one of the book's authors, R. Lerdorf.

1. D. Bradstock et al., *DB2 SQL Procedure Language for Linux, Unix, and Windows*, IBM Press, 2005.
2. Y.-M. Chang et al., “Using Oracle PL/SQL”  
<http://infolab.stanford.edu/~ullman/fcdbl/oracle/or-plsql.html>
3. M. Fisher, J. Ellis, and J. Bruce, *JDBC API Tutorial and Reference*, Prentice-Hall, Upper Saddle River, NJ, 2003.
4. ISO/IEC Report 9075-4, 2003.
5. J. Melton, *Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM*, Morgan-Kaufmann, San Francisco, 1998.
6. Microsoft Corp., “Transact-SQL Reference”  
<http://msdn2.microsoft.com/en-us/library/ms189826.aspx>
7. K. Tatroe, R. Lerdorf, and P. MacIntyre, *Programming PHP*, O'Reilly Media, Cambridge, MA, 2006.



# Chapter 10

# Advanced Topics in Relational Databases

This chapter introduces additional topics that are of interest to the database programmer. We begin with a section on the SQL standard for authorization of access to database elements. Next, we see the SQL extension that allows for recursive programming in SQL — queries that use their own results. Then, we look at the object-relational model, and how it is implemented in the SQL standard.

The remainder of the chapter concerns “OLAP,” or on-line analytic processing. OLAP refers to complex queries of a nature that causes them to take significant time to execute. Because they are so expensive, some special technology has developed to handle them efficiently. One important direction is an implementation of relations, called the “data cube,” that is rather different from the conventional bag-of-tuples approach of SQL.

## 10.1 Security and User Authorization in SQL

SQL postulates the existence of *authorization ID*'s, which are essentially user names. SQL also has a special authorization ID called PUBLIC, which includes any user. Authorization ID's may be granted privileges, much as they would be in the file system environment maintained by an operating system. For example, a UNIX system generally controls three kinds of privileges: read, write, and execute. That list of privileges makes sense, because the protected objects of a UNIX system are files, and these three operations characterize well the things one typically does with files. However, databases are much more complex than file systems, and the kinds of privileges used in SQL are correspondingly more complex.

In this section, we shall first learn what privileges SQL allows on database elements. We shall then see how privileges may be acquired by users (by au-

thorization ID's, that is). Finally, we shall see how privileges may be taken away.

### 10.1.1 Privileges

SQL defines nine types of privileges: **SELECT**, **INSERT**, **DELETE**, **UPDATE**, **REFERENCES**, **USAGE**, **TRIGGER**, **EXECUTE**, and **UNDER**. The first four of these apply to a relation, which may be either a base table or a view. As their names imply, they give the holder of the privilege the right to query (select from) the relation, insert into the relation, delete from the relation, and update tuples of the relation, respectively.

A SQL statement cannot be executed without the privileges appropriate to that statement; e.g., a select-from-where statement requires the **SELECT** privilege on every table it accesses. We shall see how the module can get those privileges shortly. **SELECT**, **INSERT**, and **UPDATE** may also have an associated list of attributes, for instance, **SELECT(name, addr)**. If so, then only those attributes may be seen in a selection, specified in an insertion, or changed in an update, respectively. Note that, when granted, privileges such as these will be associated with a particular relation, so it will be clear at that time to what relation attributes **name** and **addr** belong.

The **REFERENCES** privilege on a relation is the right to refer to that relation in an integrity constraint. These constraints may take any of the forms mentioned in Chapter 7, such as assertions, attribute- or tuple-based checks, or referential integrity constraints. The **REFERENCES** privilege may also have an attached list of attributes, in which case only those attributes may be referenced in a constraint. A constraint cannot be created unless the owner of the schema in which the constraint appears has the **REFERENCES** privilege on all data involved in the constraint.

**USAGE** is a privilege that applies to several kinds of schema elements other than relations and assertions (see Section 9.2.2); it is the right to use that element in one's own declarations. The **TRIGGER** privilege on a relation is the right to define triggers on that relation. **EXECUTE** is the right to execute a piece of code, such as a PSM procedure or function. Finally, **UNDER** is the right to create subtypes of a given type. The matter of types appears in Section 10.4.

**Example 10.1:** Let us consider what privileges are needed to execute the insertion statement of Fig. 6.15, which we reproduce here as Fig. 10.1. First, it is an insertion into the relation **Studio**, so we require an **INSERT** privilege on **Studio**. However, since the insertion specifies only the component for attribute **name**, it is acceptable to have either the privilege **INSERT** or the privilege **INSERT(name)** on relation **Studio**. The latter privilege allows us to insert **Studio** tuples that specify only the **name** component and leave other components to take their default value or **NULL**, which is what Fig. 10.1 does.

However, notice that the insertion statement of Fig. 10.1 involves two sub-queries, starting at lines (2) and (5). To carry out these selections we require

## Triggers and Privileges

It is a bit subtle how privileges are handled for triggers. First, if you have the TRIGGER privilege for a relation, you can attempt to create any trigger you like on that relation. However, since the condition and action portions of the trigger are likely to query and/or modify portions of the database, the trigger creator must have the necessary privileges for those actions. When someone performs an activity that awakens the trigger, they do not need the privileges that the trigger condition and action require; the trigger is executed under the privileges of its creator.

```
1) INSERT INTO Studio(name)
2)     SELECT DISTINCT studioName
3)     FROM Movies
4)     WHERE studioName NOT IN
5)         (SELECT name
6)             FROM Studio);
```

Figure 10.1: Adding new studios

the privileges needed for the subqueries. Thus, we need the SELECT privilege on both relations involved in FROM clauses: `Movies` and `Studio`. Note that just because we have the INSERT privilege on `Studio` doesn't mean we have the SELECT privilege on `Studio`, or vice versa. Since it is only particular attributes of `Movies` and `Studio` that get selected, it is sufficient to have the privilege `SELECT(studioName)` on `Movies` and the privilege `SELECT(name)` on `Studio`, or privileges that include these attributes within a list of attributes. □

### 10.1.2 Creating Privileges

There are two aspects to the awarding of privileges: how they are created initially, and how they are passed from user to user. We shall discuss initialization here and the transmission of privileges in Section 10.1.4.

First, SQL elements such as schemas or modules have an owner. The owner of something has all privileges associated with that thing. There are three points at which ownership is established in SQL.

1. When a schema is created, it and all the tables and other schema elements in it are owned by the user who created it. This user thus has all possible privileges on elements of the schema.
2. When a session is initiated by a CONNECT statement, there is an opportunity to indicate the user with an AUTHORIZATION clause. For instance,

the connection statement

```
CONNECT TO Starfleet-sql-server AS conn1
AUTHORIZATION kirk;
```

would create a connection called `conn1` to a database server whose name is `Starfleet-sql-server`, on behalf of user `kirk`. Presumably, the SQL implementation would verify that the user name is valid, for example by asking for a password. It is also possible to include the password in the `AUTHORIZATION` clause, as we discussed in Section 9.2.5. That approach is somewhat insecure, since passwords are then visible to someone looking over Kirk's shoulder.

3. When a module is created, there is an option to give it an owner by using an `AUTHORIZATION` clause. For instance, a clause

```
AUTHORIZATION picard;
```

in a module-creation statement would make user `picard` the owner of the module. It is also acceptable to specify no owner for a module, in which case the module is publicly executable, but the privileges necessary for executing any operations in the module must come from some other source, such as the user associated with the connection and session during which the module is executed.

### 10.1.3 The Privilege-Checking Process

As we saw above, each module, schema, and session has an associated user; in SQL terms, there is an associated authorization ID for each. Any SQL operation has two parties:

1. The database elements upon which the operation is performed and
2. The agent that causes the operation.

The privileges available to the agent derive from a particular authorization ID called the *current authorization ID*. That ID is either

- a) The module authorization ID, if the module that the agent is executing has an authorization ID, or
- b) The session authorization ID if not.

We may execute the SQL operation only if the current authorization ID possesses all the privileges needed to carry out the operation on the database elements involved.

**Example 10.2:** To see the mechanics of checking privileges, let us reconsider Example 10.1. We might suppose that the referenced tables — `Movies` and `Studio` — are part of a schema called `MovieSchema`, which was created by and is owned by user `janeway`. At this point, user `janeway` has all privileges on these tables and any other elements of the schema `MovieSchema`. She may choose to grant some privileges to others by the mechanism to be described in Section 10.1.4, but let us assume none have been granted yet. There are several ways that the insertion of Example 10.1 can be executed.

1. The insertion could be executed as part of a module created by user `janeway` and containing an `AUTHORIZATION janeway` clause. The module authorization ID, if there is one, always becomes the current authorization ID. Then, the module and its SQL insertion statement have exactly the same privileges user `janeway` has, which includes all privileges on the tables `Movies` and `Studio`.
2. The insertion could be part of a module that has no owner. User `janeway` opens a connection with an `AUTHORIZATION janeway` clause in the `CONNECT` statement. Now, `janeway` is again the current authorization ID, so the insertion statement has all the privileges needed.
3. User `janeway` grants all privileges on tables `Movies` and `Studio` to user `archer`, or perhaps to the special user `PUBLIC`, which stands for “all users.” Suppose the insertion statement is in a module with the clause

`AUTHORIZATION archer`

Since the current authorization ID is now `archer`, and this user has the needed privileges, the insertion is again permitted.

4. As in (3), suppose user `janeway` has given user `archer` the needed privileges. Also, suppose the insertion statement is in a module without an owner; it is executed in a session whose authorization ID was set by an `AUTHORIZATION archer` clause. The current authorization ID is thus `archer`, and that ID has the needed privileges.

□

There are several principles that are illustrated by Example 10.2. We shall summarize them below.

- The needed privileges are always available if the data is owned by the same user as the user whose ID is the current authorization ID. Scenarios (1) and (2) above illustrate this point.
- The needed privileges are available if the user whose ID is the current authorization ID has been granted those privileges by the owner of the data, or if the privileges have been granted to user `PUBLIC`. Scenarios (3) and (4) illustrate this point.

- Executing a module owned by the owner of the data, or by someone who has been granted privileges on the data, makes the needed privileges available. Of course, one needs the **EXECUTE** privilege on the module itself. Scenarios (1) and (3) illustrate this point.
- Executing a publicly available module during a session whose authorization ID is that of a user with the needed privileges is another way to execute the operation legally. Scenarios (2) and (4) illustrate this point.

#### 10.1.4 Granting Privileges

So far, the only way we have seen to have privileges on a database element is to be the creator and owner of that element. SQL provides a **GRANT** statement to allow one user to give a privilege to another. The first user retains the privilege granted, as well; thus **GRANT** can be thought of as “copy a privilege.”

There is one important difference between granting privileges and copying. Each privilege has an associated *grant option*. That is, one user may have a privilege like **SELECT** on table **Movies** “with grant option,” while a second user may have the same privilege, but without the grant option. Then the first user may grant the privilege **SELECT** on **Movies** to a third user, and moreover that grant may be with or without the grant option. However, the second user, who does not have the grant option, may not grant the privilege **SELECT** on **Movies** to anyone else. If the third user got the privilege with the grant option, then that user may grant the privilege to a fourth user, again with or without the grant option, and so on.

A *grant statement* has the form:

**GRANT <privilege list> ON <database element> TO <user list>**

possibly followed by **WITH GRANT OPTION**.

The database element is typically a relation, either a base table or a view. If it is another kind of element, the name of the element is preceded by the type of that element, e.g., **ASSERTION**. The privilege list is a list of one or more privileges, e.g., **SELECT** or **INSERT(name)**. Optionally, the keywords **ALL PRIVILEGES** may appear here, as a shorthand for all the privileges that the grantor may legally grant on the database element in question.

In order to execute this grant statement legally, the user executing it must possess the privileges granted, and these privileges must be held with the grant option. However, the grantor may hold a more general privilege (with the grant option) than the privilege granted. For instance, the privilege **INSERT(name)** on table **Studio** might be granted, while the grantor holds the more general privilege **INSERT** on **Studio**, with grant option.

**Example 10.3:** User **janeway**, who is the owner of the **MovieSchema** schema that contains tables

```
Movies(title, year, length, genre, studioName, producerC#)
Studio(name, address, presC#)
```

grants the INSERT and SELECT privileges on table Studio and privilege SELECT on Movies to users kirk and picard. Moreover, she includes the grant option with these privileges. The grant statements are:

```
GRANT SELECT, INSERT ON Studio TO kirk, picard
    WITH GRANT OPTION;
GRANT SELECT ON Movies TO kirk, picard
    WITH GRANT OPTION;
```

Now, picard grants to user sisko the same privileges, but without the grant option. The statements executed by picard are:

```
GRANT SELECT, INSERT ON Studio TO sisko;
GRANT SELECT ON Movies TO sisko;
```

Also, kirk grants to sisko the minimal privileges needed for the insertion of Fig. 10.1, namely SELECT and INSERT(*name*) on Studio and SELECT on Movies. The statements are:

```
GRANT SELECT, INSERT(name) ON Studio TO sisko;
GRANT SELECT ON Movies TO sisko;
```

Note that sisko has received the SELECT privilege on Movies and Studio from two different users. He has also received the INSERT(*name*) privilege on Studio twice: directly from kirk and via the generalized privilege INSERT from picard.

## 10.1.5 Grant Diagrams

Because of the complex web of grants and overlapping privileges that may result from a sequence of grants, it is useful to represent grants by a graph called a *grant diagram*. A SQL system maintains a representation of this diagram to keep track of both privileges and their origins (in case a privilege is revoked; see Section 10.1.6).

The nodes of a grant diagram correspond to a user and a privilege. Note that the ability to do something (e.g., SELECT on relation *R*) with the grant option and the same ability without the grant option are different privileges. These two different privileges, even if they belong to the same user, must be represented by two different nodes. Likewise, a user may hold two privileges, one of which is strictly more general than the other (e.g., SELECT on *R* and SELECT on *R(A)*). These two privileges are also represented by two different nodes.

If user *U* grants privilege *P* to user *V*, and this grant was based on the fact that *U* holds privilege *Q* (*Q* could be *P* with the grant option, or it could be

some generalization of  $P$ , again with the grant option), then we draw an arc from the node for  $U/Q$  to the node for  $V/P$ . As we shall see, privileges may be lost when arcs of this graph are deleted. That is why we use separate nodes for a pair of privileges, one of which includes the other, such as a privilege with and without the grant option. If the more powerful privilege is lost, the less powerful one might still be retained.

**Example 10.4:** Figure 10.2 shows the grant diagram that results from the sequence of grant statements of Example 10.3. We use the convention that a \* after a user-privilege combination indicates that the privilege includes the grant option. Also, \*\* after a user-privilege combination indicates that the privilege derives from ownership of the database element in question and was not due to a grant of the privilege from elsewhere. This distinction will prove important when we discuss revoking privileges in Section 10.1.6. A doubly starred privilege automatically includes the grant option.  $\square$

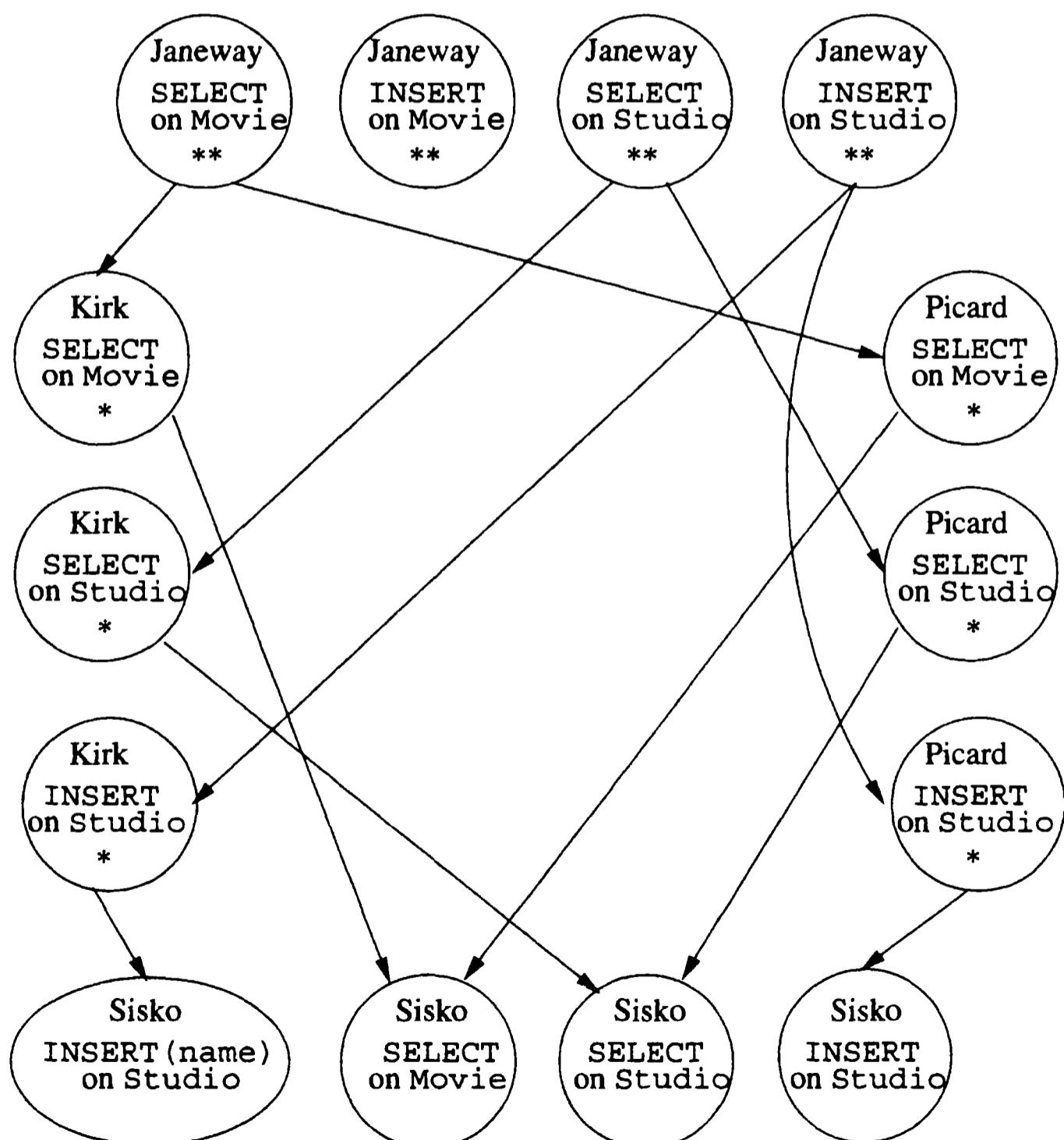


Figure 10.2: A grant diagram

### 10.1.6 Revoking Privileges

A granted privilege can be revoked at any time. The revoking of privileges may be required to *cascade*, in the sense that revoking a privilege with the grant option that has been passed on to other users may require those privileges to be revoked too. The simple form of a *revoke statement* begins:

```
REVOKE <privilege list> ON <database element> FROM <user list>
```

The statement ends with one of the following:

1. **CASCADE**. If chosen, then when the specified privileges are revoked, we also revoke any privileges that were granted *only* because of the revoked privileges. More precisely, if user  $U$  has revoked privilege  $P$  from user  $V$ , based on privilege  $Q$  belonging to  $U$ , then we delete the arc in the grant diagram from  $U/Q$  to  $V/P$ . Now, any node that is not accessible from some ownership node (doubly starred node) is also deleted.
2. **RESTRICT**. In this case, the revoke statement cannot be executed if the cascading rule described in the previous item would result in the revoking of any privileges due to the revoked privileges having been passed on to others.

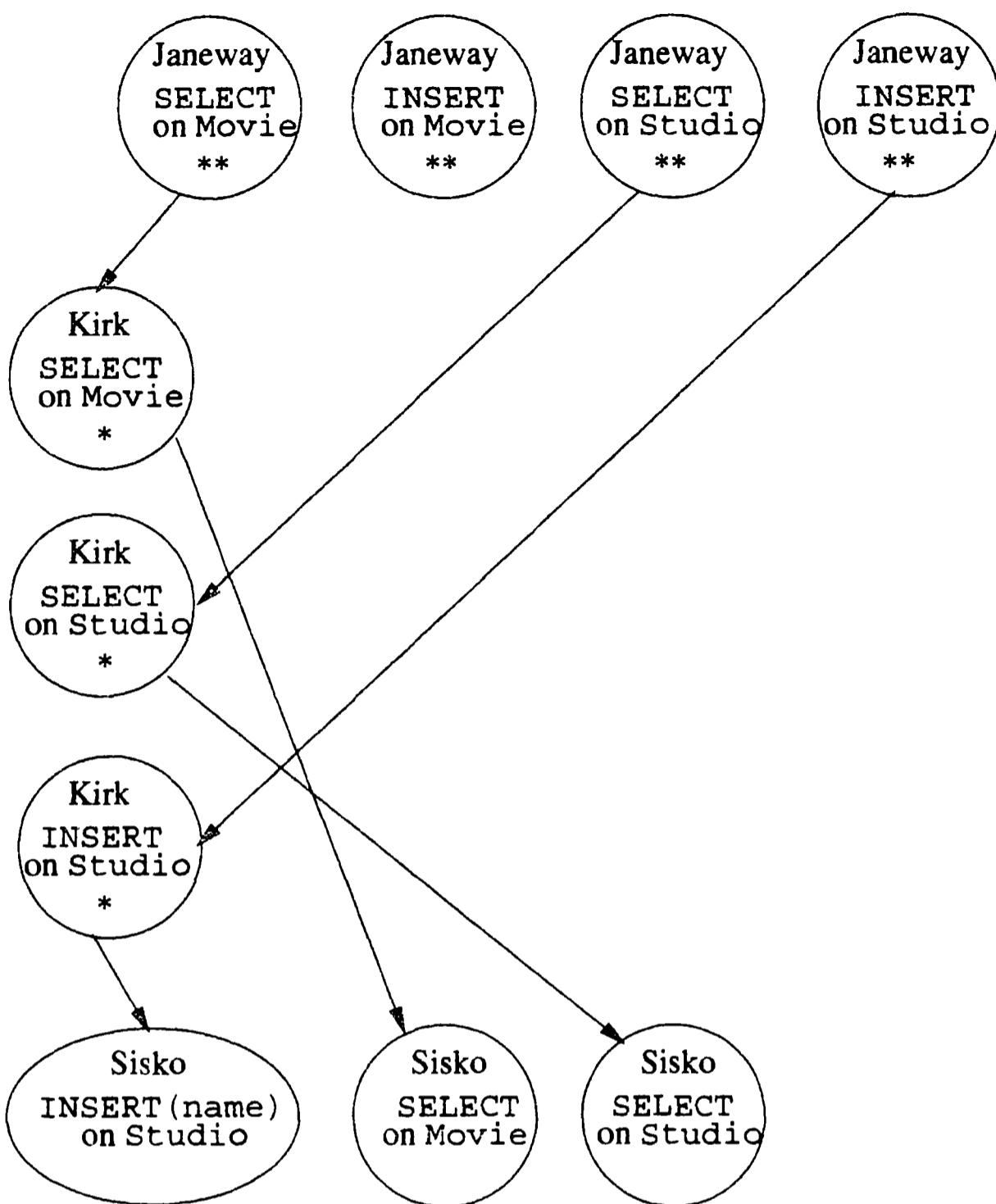
It is permissible to replace REVOKE by REVOKE GRANT OPTION FOR, in which case the core privileges themselves remain, but the option to grant them to others is removed. We may have to modify a node, redirect arcs, or create a new node to reflect the changes for the affected users. This form of REVOKE also must be followed by either CASCADE or RESTRICT.

**Example 10.5:** Continuing with Example 10.3, suppose that **janeway** revokes the privileges she granted to **picard** with the statements:

```
REVOKE SELECT, INSERT ON Studio FROM picard CASCADE;
REVOKE SELECT ON Movies FROM picard CASCADE;
```

We delete the arcs of Fig. 10.2 from these **janeway** privileges to the corresponding **picard** privileges. Since CASCADE was stipulated, we also have to see if there are any privileges that are not reachable in the graph from a doubly starred (ownership-based) privilege. Examining Fig. 10.2, we see that **picard**'s privileges are no longer reachable from a doubly starred node (they might have been, had there been another path to a **picard** node). Also, **sisko**'s privilege to **INSERT** into **Studio** is no longer reachable. We thus delete not only **picard**'s privileges from the grant diagram, but we delete **sisko**'s **INSERT** privilege.

Note that we do not delete **sisko**'s **SELECT** privileges on **Movies** and **Studio** or his **INSERT(name)** privilege on **Studio**, because these are all reachable from **janeway**'s ownership-based privileges via **kirk**'s privileges. The resulting grant diagram is shown in Fig. 10.3.  $\square$

Figure 10.3: Grant diagram after revocation of `picard`'s privileges

**Example 10.6:** There are a few subtleties that we shall illustrate with abstract examples. First, when we revoke a general privilege  $p$ , we do not also revoke a privilege that is a special case of  $p$ . For instance, consider the following sequence of steps, whereby user  $U$ , the owner of relation  $R$ , grants the `INSERT` privilege on relation  $R$  to user  $V$ , and also grants the `INSERT(A)` privilege on the same relation.

Step	By	Action
1	$U$	<code>GRANT INSERT ON R TO V</code>
2	$U$	<code>GRANT INSERT(A) ON R TO V</code>
3	$U$	<code>REVOKE INSERT ON R FROM V RESTRICT</code>

When  $U$  revokes `INSERT` from  $V$ , the `INSERT(A)` privilege remains. The grant diagrams after steps (2) and (3) are shown in Fig. 10.4.

Notice that after step (2) there are two separate nodes for the two similar but distinct privileges that user  $V$  has. Also observe that the `RESTRICT` option in step (3) does not prevent the revocation, because  $V$  had not granted the

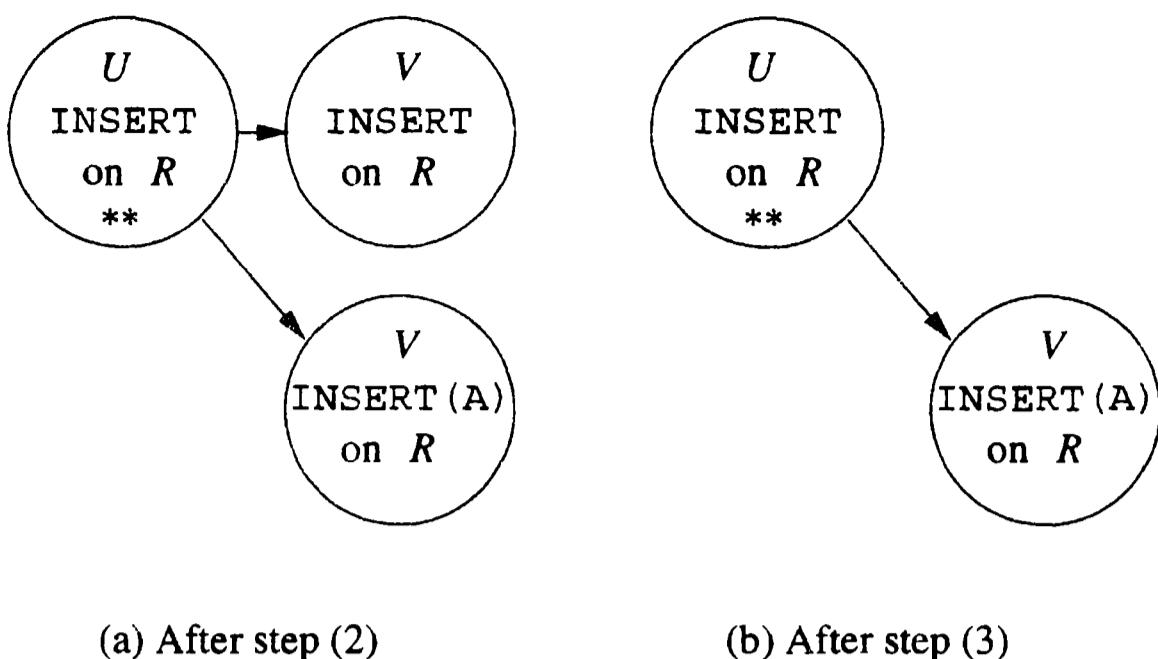


Figure 10.4: Revoking a general privilege leaves a more specific privilege

option to any other user. In fact,  $V$  could not have granted either privilege, because  $V$  obtained them without grant option.  $\square$

**Example 10.7:** Now, let us consider a similar example where  $U$  grants  $V$  a privilege  $p*$  that includes the grant option and then revokes only the grant option. Assume the grant by  $U$  was based on its privilege  $q*$ . In this case, we must replace the arc from the  $U/q*$  node to  $V/p*$  by an arc from  $U/q*$  to  $V/p$ , i.e., the same privilege without the grant option. If there was no such node  $V/p$ , it must be created. In normal circumstances, the node  $V/p*$  becomes unreachable, and any grants of  $p$  made by  $V$  will also be unreachable. However, it may be that  $V$  was granted  $p*$  by some other user besides  $U$ , in which case the  $V/p*$  node remains accessible.

Here is a typical sequence of steps:

Step	By	Action
1	$U$	GRANT $p$ TO $V$ WITH GRANT OPTION
2	$V$	GRANT $p$ TO $W$
3	$U$	REVOKE GRANT OPTION FOR $p$ FROM $V$ CASCADE

In step (1),  $U$  grants the privilege  $p$  to  $V$  with the grant option. In step (2),  $V$  uses the grant option to grant  $p$  to  $W$ . The diagram is then as shown in Fig. 10.5(a).

Then in step (3),  $U$  revokes the grant option for privilege  $p$  from  $V$ , but does not revoke the privilege itself. Since there is no node  $V/p$ , we create one. The arc from  $U/p**$  to  $V/P*$  is removed and replaced by one from  $U/p**$  to  $V/p$ .

Now, the nodes  $V/p*$  and  $W/p$  are not reachable from any  $\star\star$  node. Thus, these nodes are deleted from the diagram. The resulting grant diagram is shown in Fig. 10.5(b).  $\square$

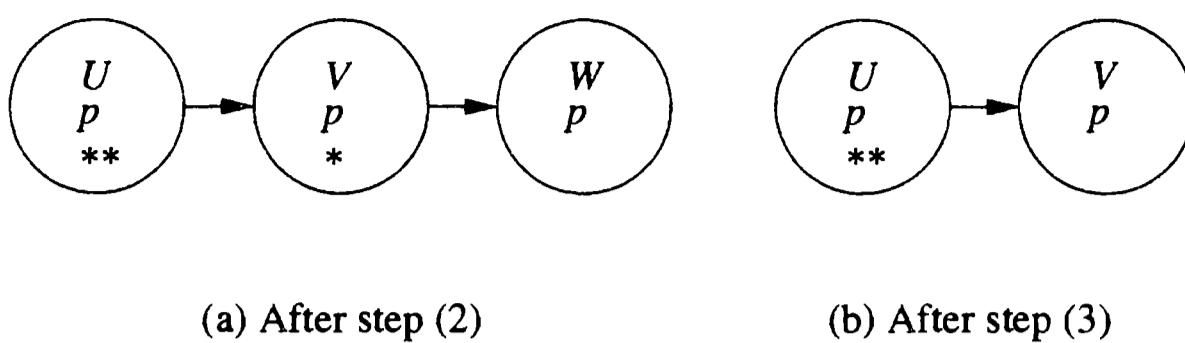


Figure 10.5: Revoking a grant option leaves the underlying privilege

### 10.1.7 Exercises for Section 10.1

**Exercise 10.1.1:** Indicate what privileges are needed to execute the following queries. In each case, mention the most specific privileges as well as general privileges that are sufficient.

- a) The query of Fig. 6.5.
  - b) The query of Fig. 6.7.
  - c) The insertion of Fig. 6.15.
  - d) The deletion of Example 6.37.
  - e) The update of Example 6.39.
  - f) The tuple-based check of Fig. 7.3.
  - g) The assertion of Example 7.11.

**Exercise 10.1.2:** Show the grant diagrams after steps (4) through (6) of the sequence of actions listed in Fig. 10.6. Assume  $A$  is the owner of the relation to which privilege  $p$  refers.

Step	By	Action
1	<i>A</i>	GRANT <i>p</i> TO <i>B</i> WITH GRANT OPTION
2	<i>A</i>	GRANT <i>p</i> TO <i>C</i>
3	<i>B</i>	GRANT <i>p</i> TO <i>D</i> WITH GRANT OPTION
4	<i>D</i>	GRANT <i>p</i> TO <i>B</i> , <i>C</i> , <i>E</i> WITH GRANT OPTION
5	<i>B</i>	REVOKE <i>p</i> FROM <i>D</i> CASCADE
6	<i>A</i>	REVOKE <i>p</i> FROM <i>C</i> CASCADE

Figure 10.6: Sequence of actions for Exercise 10.1.2

**Exercise 10.1.3:** Show the grant diagrams after steps (5) and (6) of the sequence of actions listed in Fig. 10.7. Assume  $A$  is the owner of the relation to which privilege  $p$  refers.

Step	By	Action
1	A	GRANT $p$ TO $B$ , $E$ WITH GRANT OPTION
2	B	GRANT $p$ TO $C$ WITH GRANT OPTION
3	C	GRANT $p$ TO $D$ WITH GRANT OPTION
4	E	GRANT $p$ TO $C$
5	E	GRANT $p$ TO $D$ WITH GRANT OPTION
6	A	REVOKE GRANT OPTION FOR $p$ FROM $B$ CASCADE

Figure 10.7: Sequence of actions for Exercise 10.1.3

! **Exercise 10.1.4:** Show the final grant diagram after the following steps, assuming  $A$  is the owner of the relation to which privilege  $p$  refers.

Step	By	Action
1	A	GRANT $p$ TO $B$ WITH GRANT OPTION
2	B	GRANT $p$ TO $B$ WITH GRANT OPTION
3	A	REVOKE $p$ FROM $B$ CASCADE

## 10.2 Recursion in SQL

The SQL-99 standard includes provision for recursive definitions of queries. Although this feature is not part of the “core” SQL-99 standard that every DBMS is expected to implement, at least one major system — IBM’s DB2 — does implement the SQL-99 proposal, which we describe in this section.

### 10.2.1 Defining Recursive Relations in SQL

The `WITH` statement in SQL allows us to define temporary relations, recursive or not. To define a recursive relation, the relation can be used within the `WITH` statement itself. A simple form of the `WITH` statement is:

`WITH  $R$  AS <definition of  $R$ > <query involving  $R$ >`

That is, one defines a temporary relation named  $R$ , and then uses  $R$  in some query. The temporary relation is not available outside the query that is part of the `WITH` statement.

More generally, one can define several relations after the `WITH`, separating their definitions by commas. Any of these definitions may be recursive. Several defined relations may be mutually recursive; that is, each may be defined in terms of some of the other relations, optionally including itself. However, any relation that is involved in a recursion must be preceded by the keyword `RECURSIVE`. Thus, a more general form of `WITH` statement is shown in Fig. 10.8.

**Example 10.8 :** Many examples of the use of recursion can be found in a study of paths in a graph. Figure 10.9 shows a graph representing some flights of two

WITH

```
[RECURSIVE]  $R_1$  AS <definition of  $R_1$ >,
[RECURSIVE]  $R_2$  AS <definition of  $R_2$ >,
...
[RECURSIVE]  $R_n$  AS <definition of  $R_n$ >
<query involving  $R_1, R_2, \dots, R_n$ >
```

Figure 10.8: Form of a WITH statement defining several temporary relations

hypothetical airlines — *Untried Airlines* (UA), and *Arcane Airlines* (AA) — among the cities San Francisco, Denver, Dallas, Chicago, and New York. The data of the graph can be represented by a relation

`Flights(airline, frm, to, departs, arrives)`

and the particular tuples in this table are shown in Fig. 10.9.

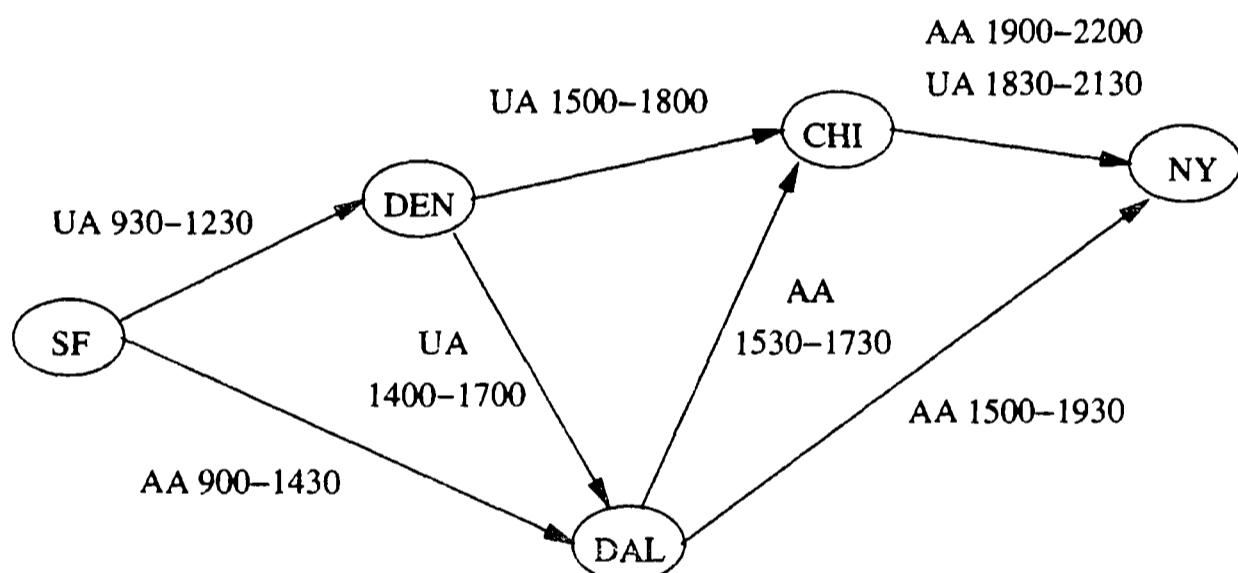


Figure 10.9: A map of some airline flights

airline	from	to	departs	arrives
UA	SF	DEN	930	1230
AA	SF	DAL	900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Figure 10.10: Tuples in the relation Flights

The simplest recursive question we can ask is “For what pairs of cities ( $x, y$ ) is it possible to get from city  $x$  to city  $y$  by taking one or more flights?” Before

writing this query in recursive SQL, it is useful to express the recursion in the Datalog notation of Section 5.3. Since many concepts involving recursion are easier to express in Datalog than in SQL, you may wish to review the terminology of that section before proceeding. The following two Datalog rules describe a relation **Reaches**( $x, y$ ) that contains exactly these pairs of cities.

1.  $\text{Reaches}(x, y) \leftarrow \text{Flights}(a, x, y, d, r)$
2.  $\text{Reaches}(x, y) \leftarrow \text{Reaches}(x, z) \text{ AND } \text{Reaches}(z, y)$

The first rule says that **Reaches** contains those pairs of cities for which there is a direct flight from the first to the second; the airline  $a$ , departure time  $d$ , and arrival time  $r$  are arbitrary in this rule. The second rule says that if you can reach from city  $x$  to city  $z$  and you can reach from  $z$  to city  $y$ , then you can reach from  $x$  to  $y$ .

Evaluating a recursive relation requires that we apply the Datalog rules repeatedly, starting by assuming there are no tuples in **Reaches**. We begin by using Rule (1) to get the following pairs in **Reaches**: (SF, DEN), (SF, DAL), (DEN, CHI), (DEN, DAL), (DAL, CHI), (DAL, NY), and (CHI, NY). These are the seven pairs represented by arcs in Fig. 10.9.

In the next round, we apply the recursive Rule (2) to put together pairs of arcs such that the head of one is the tail of the next. That gives us the additional pairs (SF, CHI), (DEN, NY), and (SF, NY). The third round combines all one- and two-arc pairs together to form paths of length up to four arcs. In this particular diagram, we get no new pairs. The relation **Reaches** thus consists of the ten pairs  $(x, y)$  such that  $y$  is reachable from  $x$  in the diagram of Fig. 10.9. Because of the way we drew the diagram, these pairs happen to be exactly those  $(x, y)$  such that  $y$  is to the right of  $x$  in Fig 10.9.

From the two Datalog rules for **Reaches** in Example 10.8, we can develop a SQL query that produces the relation **Reaches**. This SQL query places the Datalog rules for **Reaches** in a **WITH** statement, and follows it by a query. In our example, the desired result was the entire **Reaches** relation, but we could also ask some query about **Reaches**, for instance the set of cities reachable from Denver.

```

1) WITH RECURSIVE Reaches(frm, to) AS
2)           (SELECT frm, to FROM Flights)
3)           UNION
4)           (SELECT R1.frm, R2.to
5)             FROM Reaches R1, Reaches R2
6)             WHERE R1.to = R2.frm)
7) SELECT * FROM Reaches;

```

Figure 10.11: Recursive SQL query for pairs of reachable cities

Figure 10.11 shows how to express **Reaches** as a SQL query. Line (1) introduces the definition of **Reaches**, while the actual definition of this relation is in

## Mutual Recursion

There is a graph-theoretic way to check whether two relations or predicates are mutually recursive. Construct a *dependency graph* whose nodes correspond to the relations (or predicates if we are using Datalog rules). Draw an arc from relation  $A$  to relation  $B$  if the definition of  $B$  depends directly on the definition of  $A$ . That is, if Datalog is being used, then  $A$  appears in the body of a rule with  $B$  at the head. In SQL,  $A$  would appear in a `FROM` clause, somewhere in the definition of  $B$ , possibly in a subquery. If there is a cycle involving nodes  $R$  and  $S$ , then  $R$  and  $S$  are *mutually recursive*. The most common case will be a loop from  $R$  to  $R$ , indicating that  $R$  depends recursively upon itself.

lines (2) through (6).

That definition is a union of two queries, corresponding to the two Datalog rules by which `Reaches` was defined. Line (2) is the first term of the union and corresponds to the first, or basis rule. It says that for every tuple in the `Flights` relation, the second and third components (the `frm` and `to` components) are a tuple in `Reaches`.

Lines (4) through (6) correspond to Rule (2), the recursive rule, in the definition of `Reaches`. The two `Reaches` subgoals in Rule (2) are represented in the `FROM` clause by two aliases  $R1$  and  $R2$  for `Reaches`. The first component of  $R1$  corresponds to  $x$  in Rule (2), and the second component of  $R2$  corresponds to  $y$ . Variable  $z$  is represented by both the second component of  $R1$  and the first component of  $R2$ ; note that these components are equated in line (6).

Finally, line (7) describes the relation produced by the entire query. It is a copy of the `Reaches` relation. As an alternative, we could replace line (7) by a more complex query. For instance,

7) `SELECT to FROM Reaches WHERE frm = 'DEN';`

would produce all those cities reachable from Denver.  $\square$

### 10.2.2 Problematic Expressions in Recursive SQL

The SQL standard for recursion does not allow an arbitrary collection of mutually recursive relations to be written in a `WITH` clause. There is a small matter that the standard requires only that *linear* recursion be supported. A linear recursion, in Datalog terms, is one in which no rule has more than one subgoal that is mutually recursive with the head. Notice that Rule (2) in Example 10.8 has two subgoals with predicate `Reaches` that are mutually recursive with the head (a predicate is always mutually recursive with itself; see the box on Mutual

Recursion). Thus, technically, a DBMS might refuse to execute Fig. 10.11 and yet conform to the standard.<sup>1</sup>

But there is a more important restriction on SQL recursions, one that, if violated leads to recursions that cannot be executed by the query processor in any meaningful way. To be a legal SQL recursion, the definition of a recursive relation  $R$  may involve only the use of a mutually recursive relation  $S$  (including  $R$  itself) if that use is “monotone” in  $S$ . A use of  $S$  is *monotone* if adding an arbitrary tuple to  $S$  might add one or more tuples to  $R$ , or it might leave  $R$  unchanged, but it can never cause any tuple to be deleted from  $R$ . The following example suggests what can happen if the monotonicity requirement is not respected.

**Example 10.9:** Suppose relation  $R$  is a unary (one-attribute) relation, and its only tuple is  $(0)$ .  $R$  is used as an EDB relation in the following Datalog rules:

1.  $P(x) \leftarrow R(x) \text{ AND NOT } Q(x)$
2.  $Q(x) \leftarrow R(x) \text{ AND NOT } P(x)$

Informally, the two rules tell us that an element  $x$  in  $R$  is either in  $P$  or in  $Q$  but not both. Notice that  $P$  and  $Q$  are mutually recursive.

If we start out, assuming that both  $P$  and  $Q$  are empty, and apply the rules once, we find that  $P = \{(0)\}$  and  $Q = \{(0)\}$ ; that is,  $(0)$  is in both IDB relations. On the next round, we apply the rules to the new values for  $P$  and  $Q$  again, and we find that now both are empty. This cycle repeats as long as we like, but we never converge to a solution.

In fact, there are two “solutions” to the Datalog rules:

- a)  $P = \{(0)\} \quad Q = \emptyset$
- b)  $P = \emptyset \quad Q = \{(0)\}$

However, there is no reason to assume one over the other, and the simple iteration we suggested as a way to compute recursive relations never converges to either. Thus, we cannot answer a simple question such as “Is  $P(0)$  true?”

The problem is not restricted to Datalog. The two Datalog rules of this example can be expressed in recursive SQL. Figure 10.12 shows one way of doing so. This SQL does not adhere to the standard, and no DBMS should execute it.  $\square$

The problem in Example 10.9 is that the definitions of  $P$  and  $Q$  in Fig. 10.12 are not monotone. Look at the definition of  $P$  in lines (2) through (5) for instance.  $P$  depends on  $Q$ , with which it is mutually recursive, but adding a tuple to  $Q$  can delete a tuple from  $P$ . Notice that if  $R = \{(0)\}$  and  $Q$  is empty, then  $P = \{(0)\}$ . But if we add  $(0)$  to  $Q$ , then we delete  $(0)$  from  $P$ . Thus, the definition of  $P$  is not monotone in  $Q$ , and the SQL code of Fig. 10.12 does not meet the standard.

---

<sup>1</sup>Note, however, that we can replace either one of the uses of `Reaches` in line (5) of Fig. 10.11 by `Flights`, and thus make the recursion linear. Nonlinear recursions can frequently — although not always — be made linear in this fashion.

```

1) WITH
2)     RECURSIVE P(x) AS
3)         (SELECT * FROM R)
4)     EXCEPT
5)         (SELECT * FROM Q),
.
6)     RECURSIVE Q(x) AS
7)         (SELECT * FROM R)
8)     EXCEPT
9)         (SELECT * FROM P)
.
10)    SELECT * FROM P;

```

Figure 10.12: Query with nonmonotonic behavior, illegal in SQL

**Example 10.10:** Aggregation can also lead to nonmonotonicity. Suppose we have unary (one-attribute) relations  $P$  and  $Q$  defined by the following two conditions:

1.  $P$  is the union of  $Q$  and an EDB relation  $R$ .
2.  $Q$  has one tuple that is the sum of the members of  $P$ .

We can express these conditions by a **WITH** statement, although this statement violates the monotonicity requirement of SQL. The query shown in Fig. 10.13 asks for the value of  $P$ .

```

1) WITH
2)     RECURSIVE P(x) AS
3)         (SELECT * FROM R)
4)     UNION
5)         (SELECT * FROM Q),
.
6)     RECURSIVE Q(x) AS
7)         SELECT SUM(x) FROM P
.
8)    SELECT * FROM P;

```

Figure 10.13: Nonmonotone query involving aggregation, illegal in SQL

Suppose that  $R$  consists of the tuples (12) and (34), and initially  $P$  and  $Q$  are both empty. Figure 10.14 summarizes the values computed in the first six rounds. Note that both relations are computed, in one round, from the values of the relations at the previous round. Thus,  $P$  is computed in the first round

Round	$P$	$Q$
1)	$\{(12), (34)\}$	$\{\text{NULL}\}$
2)	$\{(12), (34), \text{NULL}\}$	$\{(46)\}$
3)	$\{(12), (34), (46)\}$	$\{(46)\}$
4)	$\{(12), (34), (46)\}$	$\{(92)\}$
5)	$\{(12), (34), (92)\}$	$\{(92)\}$
6)	$\{(12), (34), (92)\}$	$\{(138)\}$

Figure 10.14: Iterative calculation for a nonmonotone aggregation

to be the same as  $R$ , and  $Q$  is  $\{\text{NULL}\}$ , since the old, empty value of  $P$  is used in line (7).

At the second round, the union of lines (3) through (5) is the set

$$R \cup \{\text{NULL}\} = \{(12), (34), \text{NULL}\}$$

so that set becomes the new value of  $P$ . The old value of  $P$  was  $\{(12), (34)\}$ , so on the second round  $Q = \{(46)\}$ . That is, 46 is the sum of 12 and 34.

At the third round, we get  $P = \{(12), (34), (46)\}$  at lines (2) through (5). Using the old value of  $P$ ,  $\{(12), (34), \text{NULL}\}$ ,  $Q$  is defined by lines (6) and (7) to be  $\{(46)\}$  again. Remember that  $\text{NULL}$  is ignored in a sum.

At the fourth round,  $P$  has the same value,  $\{(12), (34), (46)\}$ , but  $Q$  gets the value  $\{(92)\}$ , since  $12+34+46=92$ . Notice that  $Q$  has lost the tuple (46), although it gained the tuple (92). That is, adding the tuple (46) to  $P$  has caused a tuple (by coincidence the same tuple) to be deleted from  $Q$ . That behavior is the nonmonotonicity that SQL prohibits in recursive definitions, confirming that the query of Fig. 10.13 is illegal. In general, at the  $i$ th round,  $P$  will consist of the tuples (12), (34), and  $(46i - 46)$ , while  $Q$  consists only of the tuple (46).  $\square$

### 10.2.3 Exercises for Section 10.2

#### Exercise 10.2.1: The relation

`Flights(airline, frm, to, departs, arrives)`

from Example 10.8 has arrival- and departure-time information that we did not consider. Suppose we are interested not only in whether it is possible to reach one city from another, but whether the journey has reasonable connections. That is, when using more than one flight, each flight must arrive at least an hour before the next flight departs. You may assume that no journey takes place over more than one day, so it is not necessary to worry about arrival close to midnight followed by a departure early in the morning.

- a) Write this recursion in Datalog.

- b) Write the recursion in SQL.

**! Exercise 10.2.2:** In Example 10.8 we used `frm` as an attribute name. Why did we not use the more obvious name `from`?

**Exercise 10.2.3:** Suppose we have a relation

`SequelOf(movie, sequel)`

that gives the immediate sequels of a movie, of which there can be more than one. We want to define a recursive relation `FollowOn` whose pairs  $(x, y)$  are movies such that  $y$  was either a sequel of  $x$ , a sequel of a sequel, or so on.

- a) Write the definition of `FollowOn` as recursive Datalog rules.
- b) Write the definition of `FollowOn` as a SQL recursion.
- c) Write a recursive SQL query that returns the set of pairs  $(x, y)$  such that movie  $y$  is a follow-on to movie  $x$ , but is not a sequel of  $x$ .
- d) Write a recursive SQL query that returns the set of pairs  $(x, y)$  meaning that  $y$  is a follow-on of  $x$ , but is neither a sequel nor a sequel of a sequel.
- ! e)** Write a recursive SQL query that returns the set of movies  $x$  that have at least two follow-ons. Note that both could be sequels, rather than one being a sequel and the other a sequel of a sequel.
- ! f)** Write a recursive SQL query that returns the set of pairs  $(x, y)$  such that movie  $y$  is a follow-on of  $x$  but  $y$  has at most one follow-on.

**Exercise 10.2.4:** Suppose we have a relation

`Rel(class, rclass, mult)`

that describes how one ODL class is related to other classes. Specifically, this relation has tuple  $(c, d, m)$  if there is a relation from class  $c$  to class  $d$ . This relation is multivalued if  $m = \text{'multi'}$  and it is single-valued if  $m = \text{'single'}$ . It is possible to view `Rel` as defining a graph whose nodes are classes and in which there is an arc from  $c$  to  $d$  labeled  $m$  if and only if  $(c, d, m)$  is a tuple of `Rel`. Write a recursive SQL query that produces the set of pairs  $(c, d)$  such that:

- a) There is a path from class  $c$  to class  $d$  in the graph described above.
- b) There is a path from  $c$  to  $d$  along which every arc is labeled `single`.
- ! c)** There is a path from  $c$  to  $d$  along which at least one arc is labeled `multi`.
- d) There is a path from  $c$  to  $d$  but no path along which all arcs are labeled `single`.

- ! e) There is a path from  $c$  to  $d$  along which arc labels alternate `single` and `multi`.
- f) There are paths from  $c$  to  $d$  and from  $d$  to  $c$  along which every arc is labeled `single`.

## 10.3 The Object-Relational Model

The relational model and the object-oriented model typified by ODL are two important points in a spectrum of options that could underlie a DBMS. For an extended period, the relational model was dominant in the commercial DBMS world. Object-oriented DBMS's made limited inroads during the 1990's, but never succeeded in winning significant market share from the vendors of relational DBMS's. Rather, the vendors of relational systems have moved to incorporate many of the ideas found in ODL or other object-oriented-database proposals. As a result, many DBMS products that used to be called "relational" are now called "object-relational."

This section extends the abstract relational model to incorporate several important object-relational ideas. It is followed by sections that cover object-relational extensions of SQL. We introduce the concept of object-relations in Section 10.3.1, then discuss one of its earliest embodiments --- nested relations --- in Section 10.3.2. ODL-like references for object-relations are discussed in Section 10.3.3, and in Section 10.3.4 we compare the object-relational model with the pure object-oriented approach.

### 10.3.1 From Relations to Object-Relations

While the relation remains the fundamental concept, the relational model has been extended to the *object-relational model* by incorporation of features such as:

1. *Structured types for attributes.* Instead of allowing only atomic types for attributes, object-relational systems support a type system like ODL's: types built from atomic types and type constructors for structs, sets, and bags, for instance. Especially important is a type that is a bag of structs, which is essentially a relation. That is, a value of one component of a tuple can be an entire relation, called a "nested relation."
2. *Methods.* These are similar to methods in ODL or any object-oriented programming system.
3. *Identifiers for tuples.* In object-relational systems, tuples play the role of objects. It therefore becomes useful in some situations for each tuple to have a unique ID that distinguishes it from other tuples, even from tuples that have the same values in all components. This ID, like the object-identifier assumed in ODL, is generally invisible to the user, although

there are even some circumstances where users can see the identifier for a tuple in an object-relational system.

4. *References.* While the pure relational model has no notion of references or pointers to tuples, object-relational systems can use these references in various ways.

In the next sections, we shall elaborate upon and illustrate each of these additional capabilities of object-relational systems.

### 10.3.2 Nested Relations

In the *nested-relational model*, we allow attributes of relations to have a type that is not atomic; in particular, a type can be a relation schema. As a result, there is a convenient, recursive definition of the types of attributes and the types (schemas) of relations:

**BASIS:** An atomic type (integer, real, string, etc.) can be the type of an attribute.

**INDUCTION:** A relation's type can be any *schema* consisting of names for one or more attributes, and any legal type for each attribute. In addition, a schema also can be the type of any attribute.

In what follows, we shall generally omit atomic types where they do not matter. An attribute that is a schema will be represented by the attribute name and a parenthesized list of the attributes of its schema. Since those attributes may themselves have structure, parentheses can be nested to any depth.

**Example 10.11:** Let us design a nested-relation schema for stars that incorporates within the relation an attribute **movies**, which will be a relation representing all the movies in which the star has appeared. The relation schema for attribute **movies** will include the title, year, and length of the movie. The relation schema for the relation **Stars** will include the name, address, and birthdate, as well as the information found in **movies**. Additionally, the **address** attribute will have a relation type with attributes **street** and **city**. We can record in this relation several addresses for the star. The schema for **Stars** can be written:

```
Stars(name, address(street, city), birthdate,
      movies(title, year, length))
```

An example of a possible relation for nested relation **Stars** is shown in Fig. 10.15. We see in this relation two tuples, one for Carrie Fisher and one for Mark Hamill. The values of components are abbreviated to conserve space, and the dashed lines separating tuples are only for convenience and have no notational significance.

<i>name</i>	<i>address</i>	<i>birthdate</i>	<i>movies</i>																		
Fisher	<table border="1"> <thead> <tr> <th><i>street</i></th><th><i>city</i></th></tr> </thead> <tbody> <tr> <td>Maple</td><td>H' wood</td></tr> <tr> <td>Locust</td><td>Malibu</td></tr> </tbody> </table>	<i>street</i>	<i>city</i>	Maple	H' wood	Locust	Malibu	9/9/99	<table border="1"> <thead> <tr> <th><i>title</i></th><th><i>year</i></th><th><i>length</i></th></tr> </thead> <tbody> <tr> <td>Star Wars</td><td>1977</td><td>124</td></tr> <tr> <td>Empire</td><td>1980</td><td>127</td></tr> <tr> <td>Return</td><td>1983</td><td>133</td></tr> </tbody> </table>	<i>title</i>	<i>year</i>	<i>length</i>	Star Wars	1977	124	Empire	1980	127	Return	1983	133
<i>street</i>	<i>city</i>																				
Maple	H' wood																				
Locust	Malibu																				
<i>title</i>	<i>year</i>	<i>length</i>																			
Star Wars	1977	124																			
Empire	1980	127																			
Return	1983	133																			
Hamill	<table border="1"> <thead> <tr> <th><i>street</i></th><th><i>city</i></th></tr> </thead> <tbody> <tr> <td>Oak</td><td>B' wood</td></tr> </tbody> </table>	<i>street</i>	<i>city</i>	Oak	B' wood	8/8/88	<table border="1"> <thead> <tr> <th><i>title</i></th><th><i>year</i></th><th><i>length</i></th></tr> </thead> <tbody> <tr> <td>Star Wars</td><td>1977</td><td>124</td></tr> <tr> <td>Empire</td><td>1980</td><td>127</td></tr> <tr> <td>Return</td><td>1983</td><td>133</td></tr> </tbody> </table>	<i>title</i>	<i>year</i>	<i>length</i>	Star Wars	1977	124	Empire	1980	127	Return	1983	133		
<i>street</i>	<i>city</i>																				
Oak	B' wood																				
<i>title</i>	<i>year</i>	<i>length</i>																			
Star Wars	1977	124																			
Empire	1980	127																			
Return	1983	133																			

Figure 10.15: A nested relation for stars and their movies

In the Carrie Fisher tuple, we see her name, an atomic value, followed by a relation for the value of the address component. That relation has two attributes, **street** and **city**, and there are two tuples, corresponding to her two houses. Next comes the birthdate, another atomic value. Finally, there is a component for the **movies** attribute; this attribute has a relation schema as its type, with components for the title, year, and length of a movie. The relation for the **movies** component of the Carrie Fisher tuple has tuples for her three best-known movies.

The second tuple, for Mark Hamill, has the same components. His relation for **address** has only one tuple, because in our imaginary data, he has only one house. His relation for **movies** looks just like Carrie Fisher's because their best-known movies happen, by coincidence, to be the same. Note that these two relations are two different tuple-components. These components happen to be identical, just like two components that happened to have the same integer value, e.g., 124. □

### 10.3.3 References

The fact that movies like *Star Wars* will appear in several relations that are values of the **movies** attribute in the nested relation **Stars** is a cause of redundancy. In effect, the schema of Example 10.11 has the nested-relation analog of not being in BCNF. However, decomposing this **Stars** relation will not eliminate the redundancy. Rather, we need to arrange that among all the tuples of all the **movies** relations, a movie appears only once.

To cure the problem, object-relations need the ability for one tuple *t* to refer to another tuple *s*, rather than incorporating *s* directly in *t*. We thus add to our model an additional inductive rule: the type of an attribute also can be a

reference to a tuple with a given schema or a set of references to tuples with a given schema.

If an attribute  $A$  has a type that is a reference to a single tuple with a relation schema named  $R$ , we show the attribute  $A$  in a schema as  $A(*R)$ . Notice that this situation is analogous to an ODL relationship  $A$  whose type is  $R$ ; i.e., it connects to a single object of type  $R$ . Similarly, if an attribute  $A$  has a type that is a set of references to tuples of schema  $R$ , then  $A$  will be shown in a schema as  $A(\{*R\})$ . This situation resembles an ODL relationship  $A$  that has type  $\text{Set}\langle R \rangle$ .

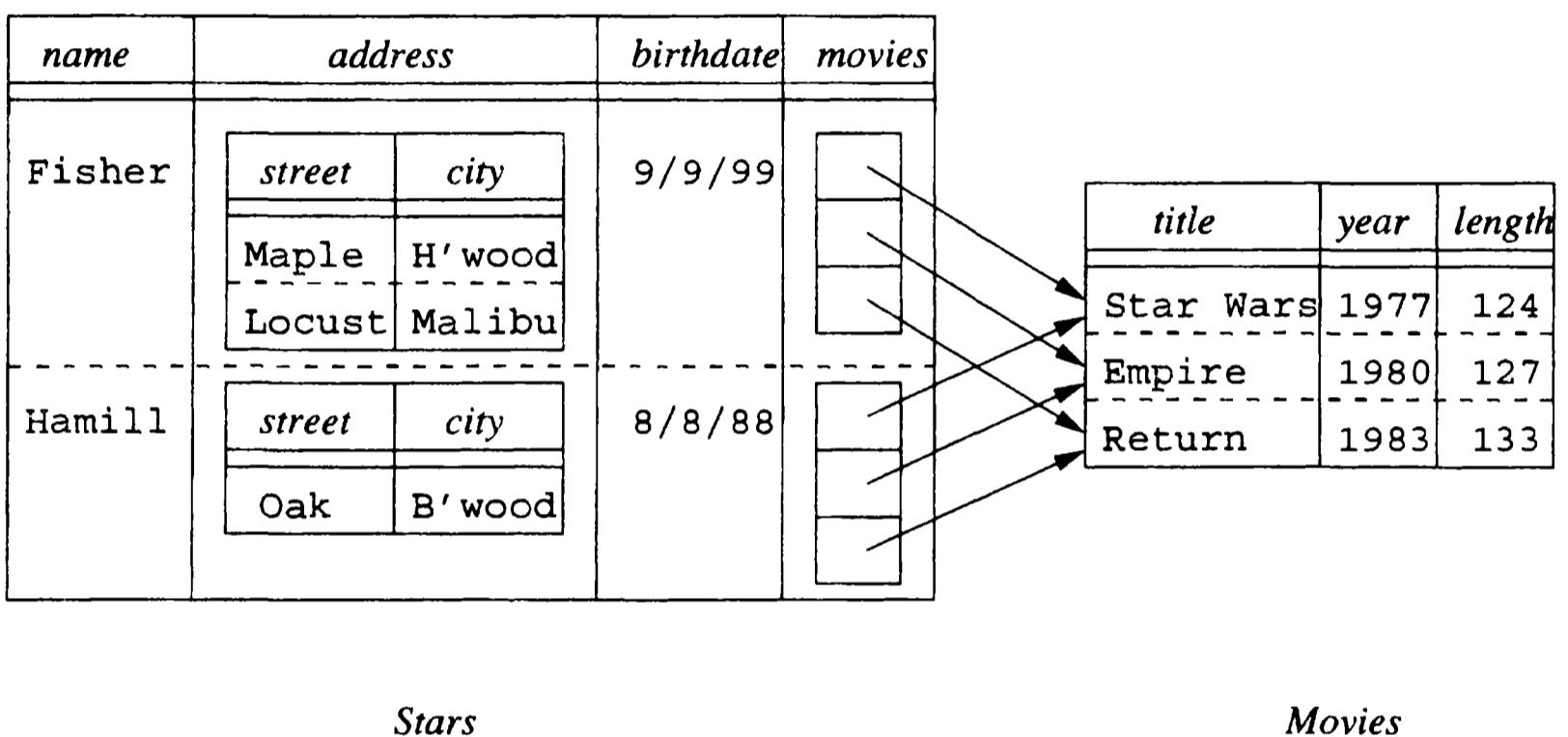


Figure 10.16: Sets of references as the value of an attribute

**Example 10.12:** An appropriate way to fix the redundancy in Fig. 10.15 is to use two relations, one for stars and one for movies. In this example only, we shall use a relation called `Movies` that is an ordinary relation with the same schema as the attribute `movies` in Example 10.11. A new relation `Stars` has a schema similar to the nested relation `Stars` of that example, but the `movies` attribute will have a type that is a set of references to `Movies` tuples. The schemas of the two relations are thus:

```

Movies(title, year, length)
Stars(name, address(street, city), birthdate,
      movies(\{*Movies\}))
```

The data of Fig. 10.15, converted to this new schema, is shown in Fig. 10.16. Notice that, because each movie has only one tuple, although it can have many references, we have eliminated the redundancy inherent in the schema of Example 10.11.  $\square$

### 10.3.4 Object-Oriented Versus Object-Relational

The object-oriented data model, as typified by ODL, and the object-relational model discussed here, are remarkably similar. Some of the salient points of comparison follow.

#### Objects and Tuples

An object's value is really a struct with components for its attributes and relationships. It is not specified in the ODL standard how relationships are to be represented, but we may assume that an object is connected to related objects by some collection of references. A tuple is likewise a struct, but in the conventional relational model, it has components for only the attributes. Relationships would be represented by tuples in another relation, as suggested in Section 4.5.2. However the object-relational model, by allowing sets of references to be a component of tuples, also allows relationships to be incorporated directly into the tuples that represent an "object" or entity.

#### Methods

We did not discuss the use of methods as part of an object-relational schema. However, in practice, the SQL-99 standard and all implementations of object-relational ideas allow the same ability as ODL to declare and define methods associated with any class or type.

#### Type Systems

The type systems of the object-oriented and object-relational models are quite similar. Each is based on atomic types and construction of new types by struct- and collection-type-constructors. The choice of collection types may vary, but all variants include at least sets and bags. Moreover, the set (or bag) of structs type plays a special role in both models. It is the type of classes in ODL, and the type of relations in the object-relational model.

#### References and Object-ID's

A pure object-oriented model uses object-ID's that are completely hidden from the user, and thus cannot be seen or queried. The object-relational model allows references to be part of a type, and thus it is possible under some circumstances for the user to see their values and even remember them for future use. You may regard this situation as anything from a serious bug to a stroke of genius, depending on your point of view, but in practice it appears to make little difference.

## Backwards Compatibility

With little difference in essential features of the two models, it is interesting to consider why object-relational systems have dominated the pure object-oriented systems in the marketplace. The reason, we believe, is as follows. As relational DBMS's evolved into object-relational DBMS's, the vendors were careful to maintain backwards compatibility. That is, newer versions of the system would still run the old code and accept the same schemas, should the user not care to adopt any of the object-oriented features. On the other hand, migration to a pure object-oriented DBMS would require the installations to rewrite and reorganize extensively. Thus, whatever competitive advantage could be argued for object-oriented database systems was insufficient to motivate many to make the switch.

### 10.3.5 Exercises for Section 10.3

**Exercise 10.3.1:** Using the notation developed for nested relations and relations with references, give one or more relation schemas that represent the following information. In each case, you may exercise some discretion regarding what attributes of a relation are included, but try to keep close to the attributes found in our running movie example. Also, indicate whether your schemas exhibit redundancy, and if so, what could be done to avoid it.

- a) Movies, with the usual attributes plus all their stars and the usual information about the stars.
- ! b) Studios, all the movies made by that studio, and all the stars of each movie, including all the usual attributes of studios, movies, and stars.
- c) Movies with their studio, their stars, and all the usual attributes of these.

**Exercise 10.3.2:** Represent the banking information of Exercise 4.1.1 in the object-relational model developed in this section. Make sure that it is easy, given the tuple for a customer, to find their account(s) and *also* easy, given the tuple for an account to find the customer(s) that hold that account. Also, try to avoid redundancy.

**! Exercise 10.3.3:** If the data of Exercise 10.3.2 were modified so that an account could be held by only one customer [as in Exercise 4.1.2(a)], how could your answer to Exercise 10.3.2 be simplified?

**! Exercise 10.3.4:** Render the players, teams, and fans of Exercise 4.1.3 in the object-relational model.

**! Exercise 10.3.5:** Render the genealogy of Exercise 4.1.6 in the object-relational model.

## 10.4 User-Defined Types in SQL

We now turn to the way SQL-99 incorporates many of the object-oriented features that we saw in Section 10.3. The central extension that turns the relational model into the object-relational model in SQL is the *user-defined type*, or UDT. We find UDT's used in two distinct ways:

1. A UDT can be the type of a table.
2. A UDT can be the type of an attribute belonging to some table.

### 10.4.1 Defining Types in SQL

The SQL-99 standard allows the programmer to define UDT's in several ways. The simplest is as a renaming of an existing type.

```
CREATE TYPE T AS <primitive type>;
```

renames a primitive type such as INTEGER. Its purpose is to prevent errors caused by accidental coercions among values that logically should not be compared or interchanged, even though they have the same primitive data type. An example should make the purpose clear.

**Example 10.13:** In our running movies example, there are several attributes of type INTEGER. These include `length` of `Movies`, `cert#` of `MovieExec`, and `presC#` of `Studio`. It makes sense to compare a value of `cert#` with a value of `presC#`, and we could even take a value from one of these two attributes and store it in a tuple as the value of the other attribute. However, It would not make sense to compare a movie length with the certificate number of a movie executive, or to take a `length` value from a `Movies` tuple and store it in the `cert#` attribute of a `MovieExec` tuple.

If we create types:

```
CREATE TYPE CertType AS INTEGER;
CREATE TYPE LengthType AS INTEGER;
```

then we can declare `cert#` and `presC#` to be of type `CertType` instead of `INTEGER` in their respective relation declarations, and we can declare `length` to be of type `LengthType` in the `Movies` declaration. In that case, an object-relational DBMS will intercept attempts to compare values of one type with the other, or to use a value of one type in place of the other. □

A more powerful form of UDT declaration in SQL is similar to a class declaration in ODL, with some distinctions. First, key declarations for a relation with a user-defined type are part of the table definition, not the type definition; that is, many SQL relations can be declared to have the same UDT but different keys and other constraints. Second, in SQL we do not treat relationships as properties. A relationship can be represented by a separate relation,

as was discussed in Section 4.10.5, or through references, which are covered in Section 10.4.5. This form of UDT definition is:

```
CREATE TYPE T AS (<attribute declarations>);
```

**Example 10.14:** Figure 10.17 shows two UDT's, `AddressType` and `StarType`. A tuple of type `AddressType` has two components, whose attributes are `street` and `city`. The types of these components are character strings of length 50 and 20, respectively. A tuple of type `StarType` also has two components. The first is attribute `name`, whose type is a 30-character string, and the second is `address`, whose type is itself a UDT `AddressType`, that is, a tuple with `street` and `city` components. □

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
);

CREATE TYPE StarType AS (
    name    CHAR(30),
    address AddressType
);
```

Figure 10.17: Two type definitions

## 10.4.2 Method Declarations in UDT's

The declaration of a method resembles the way a function in PSM is introduced; see Section 9.4.1. There is no analog of PSM procedures as methods. That is, every method returns a value of some type. While function declarations and definitions in PSM are combined, a method needs both a declaration, which follows the parenthesized list of attributes in the `CREATE TYPE` statement, and a separate definition, in a `CREATE METHOD` statement. The actual code for the method need not be PSM, although it could be. For example, the method body could be Java with JDBC used to access the database.

A method declaration looks like a PSM function declaration, with the keyword `METHOD` replacing `CREATE FUNCTION`. However, SQL methods typically have no arguments; they are applied to rows, just as ODL methods are applied to objects. In the definition of the method, `SELF` refers to this tuple, if necessary.

**Example 10.15:** Let us extend the definition of the type `AddressType` of Fig. 10.17 with a method `houseNumber` that extracts from the `street` component the portion devoted to the house address. For instance, if the `street`

component were '123 Maple St.', then `houseNumber` should return '123'. Exactly how `houseNumber` works is not visible in its declaration; the details are left for the definition. The revised type definition is thus shown in Fig. 10.18.

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
)
METHOD houseNumber() RETURNS CHAR(10);
```

Figure 10.18: Adding a method declaration to a UDT

We see the keyword `METHOD`, followed by the name of the method and a parenthesized list of its arguments and their types. In this case, there are no arguments, but the parentheses are still needed. Had there been arguments, they would have appeared, followed by their types, such as `(a INT, b CHAR(5))`. □

### 10.4.3 Method Definitions

Separately, we need to define the method. A simple form of method definition is:

```
CREATE METHOD <method name, arguments, and return type>
FOR <UDT name>
    <method body>
```

That is, the UDT for which the method is defined is indicated in a `FOR` clause. The method definition need not be contiguous to, or part of, the definition of the type to which it belongs.

**Example 10.16 :** For instance, we could define the method `houseNumber` from Example 10.15 as in Fig. 10.19. We have omitted the body of the method because accomplishing the intended separation of the string `string` as intended is nontrivial, even if a general-purpose host language is used. □

```
CREATE METHOD houseNumber() RETURNS CHAR(10)
FOR AddressType
BEGIN
    ...
END;
```

Figure 10.19: Defining a method

### 10.4.4 Declaring Relations with a UDT

Having declared a type, we may declare one or more relations whose tuples are of that type. The form of relation declarations is like that of Section 2.3.3, but the attribute declarations are omitted from the parenthesized list of elements, and replaced by a clause with `OF` and the name of the UDT. That is, the alternative form of a `CREATE TABLE` statement, using a UDT, is:

```
CREATE TABLE <table name> OF <UDT name>
  (<list of elements>);
```

The parenthesized list of elements can include keys, foreign keys, and tuple-based constraints. Note that all these elements are declared for a particular table, not for the UDT. Thus, there can be several tables with the same UDT as their row type, and these tables can have different constraints, and even different keys. If there are no constraints or key declarations desired for the table, then the parentheses are not needed.

**Example 10.17:** We could declare `MovieStar` to be a relation whose tuples are of type `StarType` by

```
CREATE TABLE MovieStar OF StarType (
  PRIMARY KEY (name)
);
```

As a result, table `MovieStar` has two attributes, `name` and `address`. The first attribute, `name`, is an ordinary character string, but the second, `address`, has a type that is itself a UDT, namely the type `AddressType`. Attribute `name` is a key for this relation, so it is not possible to have two tuples with the same name. □

### 10.4.5 References

The effect of object identity in object-oriented languages is obtained in SQL through the notion of a *reference*. A table may have a *reference column* that serves as the “identity” for its tuples. This column could be the primary key of the table, if there is one, or it could be a column whose values are generated and maintained unique by the DBMS, for example. We shall defer to Section 10.4.6 the matter of defining reference columns until we first see how reference types are used.

To refer to the tuples of a table with a reference column, an attribute may have as its type a reference to another type. If  $T$  is a UDT, then `REF( $T$ )` is the type of a reference to a tuple of type  $T$ . Further, the reference may be given a *scope*, which is the name of the relation whose tuples are referred to. Thus, an attribute  $A$  whose values are references to tuples in relation  $R$ , where  $R$  is a table whose type is the UDT  $T$ , would be declared by:

```

CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movies
);

```

Figure 10.20: Adding a best movie reference to `StarType`

A `REF(T) SCOPE R`

If no scope is specified, the reference can go to any relation of type *T*.

**Example 10.18:** Let us record in `MovieStar` the best movie for each star. Assume that we have declared an appropriate relation `Movies`, and that the type of this relation is the UDT `MovieType`; we shall define both `MovieType` and `Movies` later, in Fig. 10.21. Figure 10.20 is a new definition of `StarType` that includes an attribute `bestMovies` that is a reference to a movie. Now, if relation `MovieStar` is defined to have the UDT of Fig. 10.20, then each star tuple will have a component that refers to a `Movies` tuple — the star's best movie. □

#### 10.4.6 Creating Object ID's for Tables

In order to refer to rows of a table, such as `Movies` in Example 10.18, that table needs to have an “object-ID” for its tuples. Such a table is said to be *referenceable*. In a `CREATE TABLE` statement where the type of the table is a UDT (as in Section 10.4.4), we may include an element of the form:

`REF IS <attribute name> <how generated>`

The attribute name is a name given to the column that will serve as the object-ID for tuples. The “how generated” clause can be:

1. `SYSTEM GENERATED`, meaning that the DBMS is responsible for maintaining a unique value in this column of each tuple, or
2. `DERIVED`, meaning that the DBMS will use the primary key of the relation to produce unique values for this column.

**Example 10.19:** Figure 10.21 shows how the UDT `MovieType` and relation `Movies` could be declared so that `Movies` is referenceable. The UDT is declared in lines (1) through (4). Then the relation `Movies` is defined to have this type in lines (5) through (7). Notice that we have declared `title` and `year`, together, to be the key for relation `Movies` in line (7).

We see in line (6) that the name of the “identity” column for `Movies` is `movieID`. This attribute, which automatically becomes a fourth attribute of

```
1) CREATE TYPE MovieType AS (
2)     title    CHAR(30),
3)     year     INTEGER,
4)     genre    CHAR(10)
5) );
6)
7) CREATE TABLE Movies OF MovieType (
8)     REF IS movieID SYSTEM GENERATED,
9)     PRIMARY KEY (title, year)
10);
11)
```

Figure 10.21: Creating a referenceable table

**Movies**, along with `title`, `year`, and `genre`, may be used in queries like any other attribute of **Movies**.

Line (6) also says that the DBMS is responsible for generating the value of `movieID` each time a new tuple is inserted into `Movies`. Had we replaced `SYSTEM GENERATED` by `DERIVED`, then new tuples would get their value of `movieID` by some calculation, performed by the system, on the values of the primary-key attributes `title` and `year` taken from the new tuple. □

**Example 10.20:** Now, let us see how to represent the many-many relationship between movies and stars using references. Previously, we represented this relationship by a relation like `StarsIn` that contains tuples with the keys of `Movies` and `MovieStar`. As an alternative, we may define `StarsIn` to have references to tuples from these two relations.

First, we need to redefine `MovieStar` so it is a referenceable table, thusly:

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED,
    PRIMARY KEY (name)
);
```

Then, we may declare the relation `StarsIn` to have two attributes, which are references, one to a movie tuple and one to a star tuple. Here is a direct definition of this relation:

```
CREATE TABLE StarsIn (
    star      REF(StarType) SCOPE MovieStar,
    movie     REF(MovieType) SCOPE Movies
);
```

Optionally, we could have defined a UDT as above, and then declared `StarsIn` to be a table of that type. □

### 10.4.7 Exercises for Section 10.4

**Exercise 10.4.1:** For our running movies example, choose type names for the attributes of each of the relations. Give attributes the same UDT if their values can reasonably be compared or exchanged, and give them different UDT's if they should not have their values compared or exchanged.

**Exercise 10.4.2:** Write type declarations for the following types:

- a) `NameType`, with components for first, middle, and last names and a title.
- b) `PersonType`, with a name of the person and references to the persons that are their mother and father. You must use the type from part (a) in your declaration.
- c) `MarriageType`, with the date of the marriage and references to the husband and wife.

**Exercise 10.4.3:** Redesign our running products database schema of Exercise 2.4.1 to use type declarations and reference attributes where appropriate. In particular, in the relations `PC`, `Laptop`, and `Printer` make the `model` attribute be a reference to the `Product` tuple for that model.

! **Exercise 10.4.4:** In Exercise 10.4.3 we suggested that model numbers in the tables `PC`, `Laptop`, and `Printer` could be references to tuples of the `Product` table. Is it also possible to make the `model` attribute in `Product` a reference to the tuple in the relation for that type of product? Why or why not?

**Exercise 10.4.5:** Redesign our running battleships database schema of Exercise 2.4.3 to use type declarations and reference attributes where appropriate. Look for many-one relationships and try to represent them using an attribute with a reference type.

## 10.5 Operations on Object-Relational Data

All appropriate SQL operations from previous chapters apply to tables that are declared with a UDT or that have attributes whose type is a UDT. There are also some entirely new operations we can use, such as reference-following. However, some familiar operations, especially those that access or modify columns whose type is a UDT, involve new syntax.

### 10.5.1 Following References

Suppose  $x$  is a value of type `REF( $T$ )`. Then  $x$  refers to some tuple  $t$  of type  $T$ . We can obtain tuple  $t$  itself, or components of  $t$ , by two means:

1. Operator  $\rightarrow$  has essentially the same meaning as this operator does in C. That is, if  $x$  is a reference to a tuple  $t$ , and  $a$  is an attribute of  $t$ , then  $x \rightarrow a$  is the value of the attribute  $a$  in tuple  $t$ .
2. The DEREF operator applies to a reference and produces the tuple referenced.

**Example 10.21:** Let us use the relation **StarsIn** from Example 10.20 to find the movies in which Brad Pitt starred. Recall that the schema is

**StarsIn(star, movie)**

where **star** and **movie** are references to tuples of **MovieStar** and **Movies**, respectively. A possible query is:

- 1) **SELECT DEREF(movie)**
- 2) **FROM StarsIn**
- 3) **WHERE star->name = 'Brad Pitt';**

In line (3), the expression **star->name** produces the value of the **name** component of the **MovieStar** tuple referred to by the **star** component of any given **StarsIn** tuple. Thus, the **WHERE** clause identifies those **StarsIn** tuples whose **star** components are references to the Brad-Pitt **MovieStar** tuple. Line (1) then produces the movie tuple referred to by the **movie** component of those tuples. All three attributes — **title**, **year**, and **genre** — will appear in the printed result.

Note that we could have replaced line (1) by:

- 1) **SELECT movie**

However, had we done so, we would have gotten a list of system-generated gibberish that serves as the internal unique identifiers for certain movie tuples. We would not see the information in the referenced tuples.  $\square$

### 10.5.2 Accessing Components of Tuples with a UDT

When we define a relation to have a UDT, the tuples must be thought of as single objects, rather than lists with components corresponding to the attributes of the UDT. As a case in point, consider the relation **Movies** declared in Fig. 10.21. This relation has UDT **MovieType**, which has three attributes: **title**, **year**, and **genre**. However, a tuple  $t$  in **Movies** has only *one* component, not three. That component is the object itself.

If we “drill down” into the object, we can extract the values of the three attributes in the type **MovieType**, as well as use any methods defined for that type. However, we have to access these attributes properly, since they are not attributes of the tuple itself. Rather, every UDT has an implicitly defined *observer method* for each attribute of that UDT. The name of the observer

method for an attribute  $x$  is  $x()$ . We apply this method as we would any other method for this UDT; we attach it with a dot to an expression that evaluates to an object of this type. Thus, if  $t$  is a variable whose value is of type  $T$ , and  $x$  is an attribute of  $T$ , then  $t.x()$  is the value of  $x$  in the tuple (object) denoted by  $t$ .

**Example 10.22 :** Let us find, from the relation **Movies** of Fig. 10.21 the year(s) of movies with title *King Kong*. Here is one way to do so:

```
SELECT m.year()
FROM Movies m
WHERE m.title() = 'King Kong';
```

Even though the tuple variable  $m$  would appear not to be needed here, we need a variable whose value is an object of type **MovieType** — the UDT for relation **Movies**. The condition of the WHERE clause compares the constant '*King Kong*' to the value of  $m.title()$ , the observer method for attribute **title** applied to a **MovieType** object  $m$ . Similarly, the value in the SELECT clause is expressed  $m.year()$ ; this expression applies the observer method for **year** to the object  $m$ .  $\square$

In practice, object-relational DBMS's do not use method syntax to extract an attribute from an object. Rather, the parentheses are dropped, and we shall do so in what follows. For instance, the query of Example 10.22 will be written:

```
SELECT m.year
FROM Movies m
WHERE m.title = 'King Kong';
```

The tuple variable  $m$  is still necessary, however.

The dot operator can be used to apply methods as well as to find attribute values within objects. These methods should have the parentheses attached, even if they take no arguments.

**Example 10.23 :** Suppose relation **MovieStar** has been declared to have UDT **StarType**, which we should recall from Example 10.14 has an attribute **address** of type **AddressType**. That type, in turn, has a method **houseNumber()**, which extracts the house number from an object of type **AddressType** (see Example 10.15). Then the query

```
SELECT MAX(s.address.houseNumber())
FROM MovieStar s
```

extracts the address component from a **StarType** object  $s$ , then applies the **houseNumber()** method to that **AddressType** object. The result returned is the largest house number of any movie star.  $\square$

### 10.5.3 Generator and Mutator Functions

In order to create data that conforms to a UDT, or to change components of objects with a UDT, we can use two kinds of methods that are created automatically, along with the observer methods, whenever a UDT is defined. These are:

1. A *generator method*. This method has the name of the type and no argument. It may be invoked without being applied to any object. That is, if  $T$  is a UDT, then  $T()$  returns an object of type  $T$ , with no values in its various components.
2. *Mutator methods*. For each attribute  $x$  of UDT  $T$ , there is a mutator method  $x(v)$ . When applied to an object of type  $T$ , it changes the  $x$  attribute of that object to have value  $v$ . Notice that the mutator and observer method for an attribute each have the name of the attribute, but differ in that the mutator has an argument.

**Example 10.24:** We shall write a PSM procedure that takes as arguments a street, a city, and a name, and inserts into the relation **MovieStar** (of type **StarType** according to Example 10.17) an object constructed from these values, using calls to the proper generator and mutator functions. Recall from Example 10.14 that objects of **StarType** have a **name** component that is a character string, but an **address** component that is itself an object of type **AddressType**. The procedure **InsertStar** is shown in Fig. 10.22.

```

1) CREATE PROCEDURE InsertStar(
2)     IN s CHAR(50),
3)     IN c CHAR(20),
4)     IN n CHAR(30)
    )
5) DECLARE newAddr AddressType;
6) DECLARE newStar StarType;

BEGIN
7)     SET newAddr = AddressType();
8)     SET newStar = StarType();
9)     newAddr.street(s);
10)    newAddr.city(c);
11)    newStar.name(n);
12)    newStar.address(newAddr);
13)    INSERT INTO MovieStar VALUES(newStar);
END;
```

Figure 10.22: Creating and storing a **StarType** object

Lines (2) through (4) introduce the arguments *s*, *c*, and *n*, which will provide values for a street, city, and star name, respectively. Lines (5) and (6) declare two local variables. Each is of one of the UDT's involved in the type for objects that exist in the relation **MovieStar**. At lines (7) and (8) we create empty objects of each of these two types.

Lines (9) and (10) put real values in the object **newAddr**; these values are taken from the procedure arguments that provide a street and a city. Line (11) similarly installs the argument *n* as the value of the **name** component in the object **newStar**. Then line (12) takes the entire **newAddr** object and makes it the value of the **address** component in **newStar**. Finally, line (13) inserts the constructed object into relation **MovieStar**. Notice that, as always, a relation that has a UDT as its type has but a single component, even if that component has several attributes, such as **name** and **address** in this example.

To insert a star into **MovieStar**, we can call procedure **InsertStar**.

```
CALL InsertStar('345 Spruce St.', 'Glendale', 'Gwyneth Paltrow');
```

is an example. □

It is much simpler to insert objects into a relation with a UDT if your DBMS provides a generator function that takes values for the attributes of the UDT and returns a suitable object. For example, if we have functions **AddressType(s, c)** and **StarType(n, a)** that return objects of the indicated types, then we can make the insertion at the end of Example 10.24 with an **INSERT** statement of a familiar form:

```
INSERT INTO MovieStar VALUES(
    StarType('Gwyneth Paltrow',
        AddressType('345 Spruce St.', 'Glendale')));
```

#### 10.5.4 Ordering Relationships on UDT's

Objects that are of some UDT are inherently abstract, in the sense that there is no way to compare two objects of the same UDT, either to test whether they are “equal” or whether one is less than another. Even two objects that have all components identical will not be considered equal unless we tell the system to regard them as equal. Similarly, there is no obvious way to sort the tuples of a relation that has a UDT unless we define a function that tells which of two objects of that UDT precedes the other.

Yet there are many SQL operations that require either an equality test or both an equality and a “less than” test. For instance, we cannot eliminate duplicates if we can't tell whether two tuples are equal. We cannot group by an attribute whose type is a UDT unless there is an equality test for that UDT. We cannot use an **ORDER BY** clause or a comparison like **<** in a **WHERE** clause unless we can compare two elements.

To specify an ordering or comparison, SQL allows us to issue a **CREATE ORDERING** statement for any UDT. There are a number of forms this statement may take, and we shall only consider the two simplest options:

1. The statement

```
CREATE ORDERING FOR T EQUALS ONLY BY STATE;
```

says that two members of UDT  $T$  are considered equal if all of their corresponding components are equal. There is no  $<$  defined on objects of UDT  $T$ .

2. The following statement

```
CREATE ORDERING FOR T
ORDERING FULL BY RELATIVE WITH F;
```

says that any of the six comparisons ( $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ , and  $\neq$ ) may be performed on objects of UDT  $T$ . To tell how objects  $x_1$  and  $x_2$  compare, we apply the function  $F$  to these objects. This function must be written so that  $F(x_1, x_2) < 0$  whenever we want to conclude that  $x_1 < x_2$ ;  $F(x_1, x_2) = 0$  means that  $x_1 = x_2$ , and  $F(x_1, x_2) > 0$  means that  $x_1 > x_2$ . If we replace “**ORDERING FULL**” with “**EQUALS ONLY**,” then  $F(x_1, x_2) = 0$  indicates that  $x_1 = x_2$ , while any other value of  $F(x_1, x_2)$  means that  $x_1 \neq x_2$ . Comparison by  $<$  is impossible in this case.

**Example 10.25 :** Let us consider a possible ordering on the UDT **StarType** from Example 10.14. If we want only an equality on objects of this UDT, we could declare:

```
CREATE ORDERING FOR StarType EQUALS ONLY BY STATE;
```

That statement says that two objects of **StarType** are equal if and only if their names are equal as character strings, and their addresses are equal as objects of UDT **AddressType**.

The problem is that, unless we define an ordering for **AddressType**, an object of that type is not even equal to itself. Thus, we also need to create at least an equality test for **AddressType**. A simple way to do so is to declare that two **AddressType** objects are equal if and only if their streets and cities are each equal as strings. We could do so by:

```
CREATE ORDERING FOR AddressType EQUALS ONLY BY STATE;
```

Alternatively, we could define a complete ordering of **AddressType** objects. One reasonable ordering is to order addresses first by cities, alphabetically, and among addresses in the same city, by street address, alphabetically. To do so, we have to define a function, say **AddrLEG**, that takes two **AddressType** arguments and returns a negative, zero, or positive value to indicate that the first is less than, equal to, or greater than the second. We declare:

```
CREATE ORDERING FOR AddressType
ORDERING FULL BY RELATIVE WITH AddrLEG;
```

The function `AddrLEG` is shown in Fig. 10.23. Notice that if we reach line (7), it must be that the two `city` components are the same, so we compare the `street` components. Likewise, if we reach line (9), the only remaining possibility is that the cities are the same and the first street precedes the second alphabetically. □

```
1) CREATE FUNCTION AddrLEG(
2)     x1 AddressType,
3)     x2 AddressType
4) ) RETURNS INTEGER

5) IF x1.city() < x2.city() THEN RETURN(-1)
6) ELSEIF x1.city() > x2.city() THEN RETURN(1)
7) ELSEIF x1.street() < x2.street() THEN RETURN(-1)
8) ELSEIF x1.street() = x2.street() THEN RETURN(0)
9) ELSE RETURN(1)
END IF;
```

Figure 10.23: A comparison function for address objects

In practice, commercial DBMS's each have their own way of allowing the user to define comparisons for a UDT. In addition to the two approaches mentioned above, some of the capabilities offered are:

- a) *Strict Object Equality.* Two objects are equal if and only if they are the same object.
- b) *Method-Defined Equality.* A function is applied to two objects and returns true or false, depending on whether or not the two objects should be considered equal.
- c) *Method-Defined Mapping.* A function is applied to one object and returns a real number. Objects are compared by comparing the real numbers returned.

## 10.5.5 Exercises for Section 10.5

**Exercise 10.5.1:** Use the `StarsIn` relation of Example 10.20 and the `Movies` and `MovieStar` relations accessible through `StarsIn` to write the following queries:

- a) Find the names of the stars of *Dogma*.

- ! b) Find the titles and years of all movies in which at least one star lives in Malibu.
- c) Find all the movies (objects of type `MovieType`) that starred Melanie Griffith.
- ! d) Find the movies (title and year) with at least five stars.

**Exercise 10.5.2:** Using your schema from Exercise 10.4.3, write the following queries. Don't forget to use references whenever appropriate.

- a) Find the manufacturers of PC's with a hard disk larger than 60 gigabytes.
- b) Find the manufacturers of laser printers.
- ! c) Produce a table giving for each model of laptop, the model of the laptop having the highest processor speed of any laptop made by the same manufacturer.

**Exercise 10.5.3:** Using your schema from Exercise 10.4.5, write the following queries. Don't forget to use references whenever appropriate and avoid joins (i.e., subqueries or more than one tuple variable in the `FROM` clause).

- a) Find the ships with a displacement of more than 35,000 tons.
- b) Find the battles in which at least one ship was sunk.
- ! c) Find the classes that had ships launched after 1930.
- !! d) Find the battles in which at least one US ship was damaged.

**Exercise 10.5.4:** Assuming the function `AddrLEG` of Fig. 10.23 is available, write a suitable function to compare objects of type `StarType`, and declare your function to be the basis of the ordering of `StarType` objects.

- ! **Exercise 10.5.5:** Write a procedure to take a star name as argument and delete from `StarsIn` and `MovieStar` all tuples involving that star.

## 10.6 On-Line Analytic Processing

An important application of databases is examination of data for patterns or trends. This activity, called *OLAP* (standing for *On-Line Analytic Processing* and pronounced “oh-lap”), generally involves highly complex queries that use one or more aggregations. These queries are often termed *OLAP queries* or *decision-support queries*. Some examples will be given in Section 10.6.2. A typical example is for a company to search for those of its products that have markedly increasing or decreasing overall sales.

Decision-support queries typically examine very large amounts of data, even if the query results are small. In contrast, common database operations, such

as bank deposits or airline reservations, each touch only a tiny portion of the database; the latter type of operation is often referred to as *OLTP* (*On-Line Transaction Processing*, spoken “oh-ell-tee-pee”).

A recent trend in DBMS’s is to provide specialized support for OLAP queries. For example, systems often support a “data cube” in some way. We shall discuss the architecture of these systems in Section 10.7.

### 10.6.1 OLAP and Data Warehouses

It is common for OLAP applications to take place in a separate copy of the master database, called a *data warehouse*. Data from many separate databases may be integrated into the warehouse. In a common scenario, the warehouse is only updated overnight, while the analysts work on a frozen copy during the day. The warehouse data thus gets out of date by as much as 24 hours, which limits the timeliness of its answers to OLAP queries, but the delay is tolerable in many decision-support applications.

There are several reasons why data warehouses play an important role in OLAP applications. First, the warehouse may be necessary to organize and centralize data in a way that supports OLAP queries; the data may initially be scattered across many different databases. But often more important is the fact that OLAP queries, being complex and touching much of the data, take too much time to be executed in a transaction-processing system with high throughput requirements. Recall the discussion of serializable transactions in Section 6.6. Trying to run a long transaction that needed to touch much of the database serializably with other transactions would stall ordinary OLTP operations more than could be tolerated. For instance, recording new sales as they occur might not be permitted if there were a concurrent OLAP query computing average sales.

### 10.6.2 OLAP Applications

A common OLAP application uses a warehouse of sales data. Major store chains will accumulate terabytes of information representing every sale of every item at every store. Queries that aggregate sales into groups and identify significant groups can be of great use to the company in predicting future problems and opportunities.

**Example 10.26:** Suppose the Aardvark Automobile Co. builds a data warehouse to analyze sales of its cars. The schema for the warehouse might be:

```
Sales(serialNo, date, dealer, price)
Autos(serialNo, model, color)
Dealers(name, city, state, phone)
```

A typical decision-support query might examine sales on or after April 1, 2006 to see how the recent average price per vehicle varies by state. Such a query is shown in Fig. 10.24.

```

SELECT state, AVG(price)
FROM Sales, Dealers
WHERE Sales.dealer = Dealers.name AND
      date >= '2006-01-04'
GROUP BY state;

```

Figure 10.24: Find average sales price by state

Notice how the query of Fig. 10.24 touches much of the data of the database, as it classifies every recent `Sales` fact by the state of the dealer that sold it. In contrast, a typical OLTP query such as “find the price at which the auto with serial number 123 was sold,” would touch only a single tuple of the data, provided there was an index on serial number. □

For another OLAP example, consider a credit-card company trying to decide whether applicants for a card are likely to be credit-worthy. The company creates a warehouse of all its current customers and their payment history. OLAP queries search for factors, such as age, income, home-ownership, and zip-code, that might help predict whether customers will pay their bills on time. Similarly, hospitals may use a warehouse of patient data — their admissions, tests administered, outcomes, diagnoses, treatments, and so on — to analyze for risks and select the best modes of treatment.

### 10.6.3 A Multidimensional View of OLAP Data

In typical OLAP applications there is a central relation or collection of data, called the *fact table*. A fact table represents events or objects of interest, such as sales in Example 10.26. Often, it helps to think of the objects in the fact table as arranged in a multidimensional space, or “cube.” Figure 10.25 suggests three-dimensional data, represented by points within the cube; we have called the dimensions car, dealer, and date, to correspond to our earlier example of automobile sales. Thus, in Fig. 10.25 we could think of each point as a sale of a single automobile, while the dimensions represent properties of that sale.

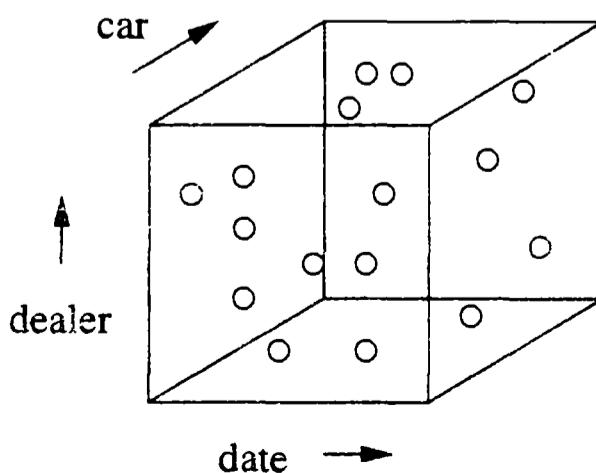


Figure 10.25: Data organized in a multidimensional space

A data space such as Fig. 10.25 will be referred to informally as a “data cube,” or more precisely as a *raw-data cube* when we want to distinguish it from the more complex “data cube” of Section 10.7. The latter, which we shall refer to as a *formal* data cube when a distinction from the raw-data cube is needed, differs from the raw-data cube in two ways:

1. It includes aggregations of the data in all subsets of dimensions, as well as the data itself.
2. Points in the formal data cube may represent an initial aggregation of points in the raw-data cube. For instance, instead of the “car” dimension representing each individual car (as we suggested for the raw-data cube), that dimension might be aggregated by model only. There are points of a formal data cube that represent the total sales of all cars of a given model by a given dealer on a given day.

The distinctions between the raw-data cube and the formal data cube are reflected in the two broad directions that have been taken by specialized systems that support cube-structured data for OLAP:

1. *ROLAP*, or *Relational OLAP*. In this approach, data may be stored in relations with a specialized structure called a “star schema,” described in Section 10.6.4. One of these relations is the “fact table,” which contains the *raw*, or unaggregated, data, and corresponds to what we called the raw-data cube. Other relations give information about the values along each dimension. The query language, index structures, and other capabilities of the system may be tailored to the assumption that data is organized this way.
2. *MOLAP*, or *Multidimensional OLAP*. Here, a specialized structure, the formal “data cube” mentioned above, is used to hold the data, including its aggregates. Nonrelational operators may be implemented by the system to support OLAP queries on data in this structure.

## 10.6.4 Star Schemas

A *star schema* consists of the schema for the fact table, which links to several other relations, called “dimension tables.” The fact table is at the center of the “star,” whose points are the dimension tables. A fact table normally has several attributes that represent *dimensions*, and one or more *dependent* attributes that represent properties of interest for the point as a whole. For instance, dimensions for sales data might include the date of the sale, the place (store) of the sale, the type of item sold, the method of payment (e.g., cash or a credit card), and so on. The dependent attribute(s) might be the sales price, the cost of the item, or the tax, for instance.

**Example 10.27 :** The **Sales** relation from Example 10.26

`Sales(serialNo, date, dealer, price)`

is a fact table. The dimensions are:

1. `serialNo`, representing the automobile sold, i.e., the position of the point in the space of possible automobiles.
2. `date`, representing the day of the sale, i.e., the position of the event in the time dimension.
3. `dealer`, representing the position of the event in the space of possible dealers.

The one dependent attribute is `price`, which is what OLAP queries to this database will typically request in an aggregation. However, queries asking for a count, rather than sum or average price would also make sense, e.g., “list the total number of sales for each dealer in the month of May, 2006.” □

Supplementing the fact table are *dimension tables* describing the values along each dimension. Typically, each dimension attribute of the fact table is a foreign key, referencing the key of the corresponding dimension table, as suggested by Fig. 10.26. The attributes of the dimension tables also describe the possible groupings that would make sense in a SQL GROUP BY query. An example should make the ideas clearer.

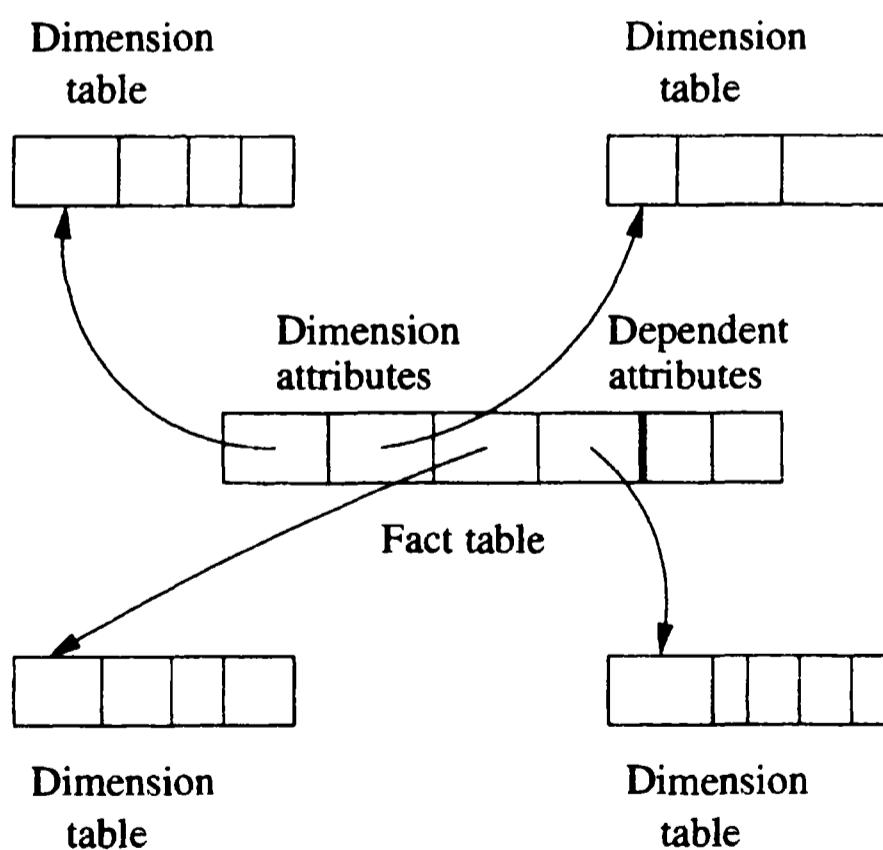


Figure 10.26: The dimension attributes in the fact table reference the keys of the dimension tables

**Example 10.28 :** For the automobile data of Example 10.26, two of the three dimension tables might be:

```
Autos(serialNo, model, color)
Dealers(name, city, state, phone)
```

Attribute `serialNo` in the fact table `Sales` is a foreign key, referencing `serialNo` of dimension table `Autos`.<sup>2</sup> The attributes `Autos.model` and `Autos.color` give properties of a given auto. If we join the fact table `Sales` with the dimension table `Autos`, then the attributes `model` and `color` may be used for grouping sales in interesting ways. For instance, we can ask for a breakdown of sales by color, or a breakdown of sales of the Gobi model by month and dealer.

Similarly, attribute `dealer` of `Sales` is a foreign key, referencing `name` of the dimension table `Dealers`. If `Sales` and `Dealers` are joined, then we have additional options for grouping our data; e.g., we can ask for a breakdown of sales by state or by city, as well as by dealer.

One might wonder where the dimension table for time (the `date` attribute of `Sales`) is. Since time is a physical property, it does not make sense to store facts about time in a database, since we cannot change the answer to questions such as “in what year does the day July 5, 2007 appear?” However, since grouping by various time units, such as weeks, months, quarters, and years, is frequently desired by analysts, it helps to build into the database a notion of time, as if there were a time “dimension table” such as

```
Days(day, week, month, year)
```

A typical tuple of this imaginary “relation” would be  $(5, 27, 7, 2007)$ , representing July 5, 2007. The interpretation is that this day is the fifth day of the seventh month of the year 2007; it also happens to fall in the 27th full week of the year 2007. There is a certain amount of redundancy, since the week is calculable from the other three attributes. However, weeks are not exactly commensurate with months, so we cannot obtain a grouping by months from a grouping by weeks, or vice versa. Thus, it makes sense to imagine that both weeks and months are represented in this “dimension table.”  $\square$

## 10.6.5 Slicing and Dicing

We can think of the points of the raw-data cube as partitioned along each dimension at some level of granularity. For example, in the time dimension, we might partition (“group by” in SQL terms) according to days, weeks, months, years, or not partition at all. For the cars dimension, we might partition by model, by color, by both model and color, or not partition. For dealers, we can partition by dealer, by city, by state, or not partition.

A choice of partition for each dimension “dices” the cube, as suggested by Fig. 10.27. The result is that the cube is divided into smaller cubes that represent groups of points whose statistics are aggregated by a query that performs this partitioning in its `GROUP BY` clause. Through the `WHERE` clause, a query also

---

<sup>2</sup>It happens that `serialNo` is also a key for the `Sales` relation, but there need not be an attribute that is both a key for the fact table and a foreign key for some dimension table.

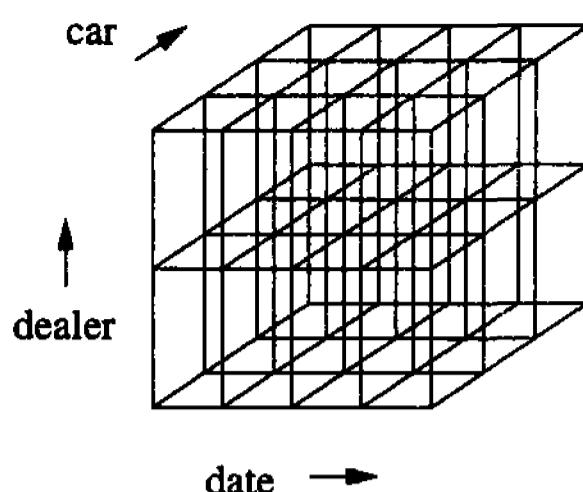


Figure 10.27: Dicing the cube by partitioning along each dimension

has the option of focusing on particular partitions along one or more dimensions (i.e., on a particular “slice” of the cube).

**Example 10.29:** Figure 10.28 suggests a query in which we ask for a slice in one dimension (the date), and dice in two other dimensions (car and dealer). The date is divided into four groups, perhaps the four years over which data has been accumulated. The shading in the diagram suggests that we are only interested in one of these years.

The cars are partitioned into three groups, perhaps sedans, SUV's, and convertibles, while the dealers are partitioned into two groups, perhaps the eastern and western regions. The result of the query is a table giving the total sales in six categories for the one year of interest. □

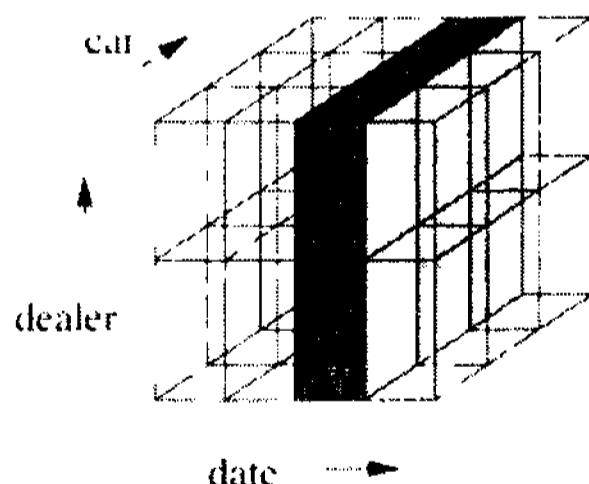


Figure 10.28: Selecting a slice of a diced cube

The general form of a so-called “slicing and dicing” query is thus:

```

SELECT <grouping attributes and aggregations>
FROM <fact table joined with some dimension tables>
WHERE <certain attributes are constant>
GROUP BY <grouping attributes>;

```

**Example 10.30:** Let us continue with our automobile example, but include the conceptual Days dimension table for time discussed in Example 10.28. If

## Drill-Down and Roll-Up

Example 10.30 illustrates two common patterns in sequences of queries that slice-and-dice the data cube.

1. *Drill-down* is the process of partitioning more finely and/or focusing on specific values in certain dimensions. Each of the steps except the last in Example 10.30 is an instance of drill-down.
2. *Roll-up* is the process of partitioning more coarsely. The last step, where we grouped by years instead of months to eliminate the effect of randomness in the data, is an example of roll-up.

the Gobi isn't selling as well as we thought it would, we might try to find out which colors are not doing well. This query uses only the **Autos** dimension table and can be written in SQL as:

```
SELECT color, SUM(price)
FROM Sales NATURAL JOIN Autos
WHERE model = 'Gobi'
GROUP BY color;
```

This query dices by color and then slices by model, focusing on a particular model, the Gobi, and ignoring other data.

Suppose the query doesn't tell us much; each color produces about the same revenue. Since the query does not partition on time, we only see the total over all time for each color. We might suppose that the recent trend is for one or more colors to have weak sales. We may thus issue a revised query that also partitions time by month. This query is:

```
SELECT color, month, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi'
GROUP BY color, month;
```

It is important to remember that the **Days** relation is not a conventional stored relation, although we may treat it as if it had the schema

**Days**(day, week, month, year)

The ability to use such a "relation" is one way that a system specialized to OLAP queries could differ from a conventional DBMS.

We might discover that red Gobis have not sold well recently. The next question we might ask is whether this problem exists at all dealers, or whether

only some dealers have had low sales of red Gobis. Thus, we further focus the query by looking at only red Gobis, and we partition along the dealer dimension as well. This query is:

```
SELECT dealer, month, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi' AND color = 'red'
GROUP BY month, dealer;
```

At this point, we find that the sales per month for red Gobis are so small that we cannot observe any trends easily. Thus, we decide that it was a mistake to partition by month. A better idea would be to partition only by years, and look at only the last two years (2006 and 2007, in this hypothetical example). The final query is shown in Fig. 10.29. □

```
SELECT dealer, year, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi' AND
      color = 'red' AND
      (year = 2006 OR year = 2007)
GROUP BY year, dealer;
```

Figure 10.29: Final slicing-and-dicing query about red Gobi sales

### 10.6.6 Exercises for Section 10.6

**Exercise 10.6.1:** An on-line seller of computers wishes to maintain data about orders. Customers can order their PC with any of several processors, a selected amount of main memory, any of several disk units, and any of several CD or DVD readers. The fact table for such a database might be:

**Orders(cust, date, proc, memory, hd, od, quant, price)**

We should understand attribute **cust** to be an ID that is the foreign key for a dimension table about customers, and understand attributes **proc**, **hd** (hard disk), and **od** (optical disk: CD or DVD, typically) analogously. For example, an **hd** ID might be elaborated in a dimension table giving the manufacturer of the disk and several disk characteristics. The **memory** attribute is simply an integer: the number of megabytes of memory ordered. The **quant** attribute is the number of machines of this type ordered by this customer, and the **price** attribute is the total cost of each machine ordered.

- a) Which are dimension attributes, and which are dependent attributes?

- b) For some of the dimension attributes, a dimension table is likely to be needed. Suggest appropriate schemas for these dimension tables.
- ! **Exercise 10.6.2:** Suppose that we want to examine the data of Exercise 10.6.1 to find trends and thus predict which components the company should order more of. Describe a series of drill-down and roll-up queries that could lead to the conclusion that customers are beginning to prefer a DVD drive to a CD drive.

## 10.7 Data Cubes

In this section, we shall consider the “formal” data cube and special operations on data presented in this form. Recall from Section 10.6.3 that the formal data cube (just “data cube” in this section) precomputes all possible aggregates in a systematic way. Surprisingly, the amount of extra storage needed is often tolerable, and as long as the warehoused data does not change, there is no penalty incurred trying to keep all the aggregates up-to-date.

In the data cube, it is normal for there to be some aggregation of the raw data of the fact table before it is entered into the data-cube and its further aggregates computed. For instance, in our cars example, the dimension we thought of as a serial number in the star schema might be replaced by the model of the car. Then, each point of the data cube becomes a description of a model, a dealer and a date, together with the sum of the sales for that model, on that date, by that dealer. We shall continue to call the points of the (formal) data cube a “fact table,” even though the interpretation of the points may be slightly different from fact tables in a star schema built from a raw-data cube.

### 10.7.1 The Cube Operator

Given a fact table  $F$ , we can define an augmented table  $\text{CUBE}(F)$  that adds an additional value, denoted  $*$ , to each dimension. The  $*$  has the intuitive meaning “any,” and it represents aggregation along the dimension in which it appears. Figure 10.30 suggests the process of adding a border to the cube in each dimension, to represent the  $*$  value and the aggregated values that it implies. In this figure we see three dimensions, with the lightest shading representing aggregates in one dimension, darker shading for aggregates over two dimensions, and the darkest cube in the corner for aggregation over all three dimensions. Notice that if the number of values along each dimension is reasonably large, then the “border” represents only a small addition to the volume of the cube (i.e., the number of tuples in the fact table). In that case, the size of the stored data  $\text{CUBE}(F)$  is not much greater than the size of  $F$  itself.

A tuple of the table  $\text{CUBE}(F)$  that has  $*$  in one or more dimensions will have for each dependent attribute the sum (or another aggregate function) of the values of that attribute in all the tuples that we can obtain by replacing

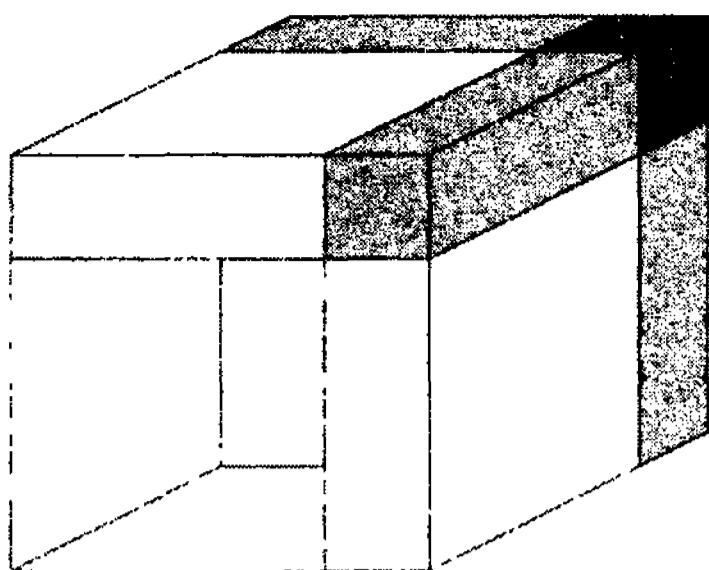


Figure 10.30: The cube operator augments a data cube with a border of aggregations in all combinations of dimensions

the \*'s by real values. In effect, we build into the data the result of aggregating along any set of dimensions. Notice, however, that the CUBE operator does not support aggregation at intermediate levels of granularity based on values in the dimension tables. For instance, we may either leave data broken down by day (or whatever the finest granularity for time is), or we may aggregate time completely, but we cannot, with the CUBE operator alone, aggregate by weeks, months, or years.

**Example 10.31:** Let us reconsider the Aardvark database from Example 10.26 in the light of what the CUBE operator can give us. Recall the fact table from that example is

```
Sales(serialNo, date, dealer, price)
```

However, the dimension represented by `serialNo` is not well suited for the cube, since the serial number is a key for `Sales`. Thus, summing the price over all dates, or over all dealers, but keeping the serial number fixed has no effect; we would still get the “sum” for the one auto with that serial number. A more useful data cube would replace the serial number by the two attributes — model and color — to which the serial number connects `Sales` via the dimension table `Autos`. Notice that if we replace `serialNo` by `model` and `color`, then the cube no longer has a key among its dimensions. Thus, an entry of the cube would have the total sales price for all automobiles of a given model, with a given color, by a given dealer, on a given date.

There is another change that is useful for the data-cube implementation of the `Sales` fact table. Since the CUBE operator normally sums dependent variables, and we might want to get average prices for sales in some category, we need both the sum of the prices for each category of automobiles (a given model of a given color sold on a given day by a given dealer) and the total number of sales in that category. Thus, the relation `Sales` to which we apply the CUBE operator is

**Sales(model, color, date, dealer, val, cnt)**

The attribute **val** is intended to be the total price of all automobiles for the given model, color, date, and dealer, while **cnt** is the total number of automobiles in that category.

Now, let us consider the relation **CUBE(Sales)**. A hypothetical tuple that would be in **CUBE(Sales)** is:

(‘Gobi’, ‘red’, ‘2001-05-21’, ‘Friendly Fred’, 45000, 2)

The interpretation is that on May 21, 2001, dealer Friendly Fred sold two red Gobis for a total of \$45,000. In **Sales**, this tuple might appear as well, or there could be in **Sales** two tuples, each with a **cnt** of 1, whose **val**’s summed to 45,000.

The tuple

(‘Gobi’, \*, ‘2001-05-21’, ‘Friendly Fred’, 152000, 7)

says that on May 21, 2001, Friendly Fred sold seven Gobis of all colors, for a total price of \$152,000. Note that this tuple is in **CUBE(Sales)** but not in **Sales**.

Relation **CUBE(Sales)** also contains tuples that represent the aggregation over more than one attribute. For instance,

(‘Gobi’, \*, ‘2001-05-21’, \*, 2348000, 100)

says that on May 21, 2001, there were 100 Gobis sold by all the dealers, and the total price of those Gobis was \$2,348,000.

(‘Gobi’, \*, \*, \*, 1339800000, 58000)

Says that over all time, dealers, and colors, 58,000 Gobis have been sold for a total price of \$1,339,800,000. Lastly, the tuple

(\*, \*, \*, \*, 3521727000, 198000)

tells us that total sales of all Aardvark models in all colors, over all time at all dealers is 198,000 cars for a total price of \$3,521,727,000. □

## 10.7.2 The Cube Operator in SQL

SQL gives us a way to apply the cube operator within queries. If we add the term **WITH CUBE** to a group-by clause, then we get not only the tuple for each group, but also the tuples that represent aggregation along one or more of the dimensions along which we have grouped. These tuples appear in the result with **NULL** where we have used **\***.

**Example 10.32:** We can construct a materialized view that is the data cube we called CUBE(Sales) in Example 10.31 by the following:

```
CREATE MATERIALIZED VIEW SalesCube AS
    SELECT model, color, date, dealer, SUM(val), SUM(cnt)
    FROM Sales
    GROUP BY model, color, date, dealer WITH CUBE;
```

The view SalesCube will then contain not only the tuples that are implied by the group-by operation, such as

(‘Gobi’, ‘red’, ‘2001-05-21’, ‘Friendly Fred’, 45000, 2)

but will also contain those tuples of CUBE(Sales) that are constructed by rolling up the dimensions listed in the GROUP BY. Some examples of such tuples would be:

```
(‘Gobi’, NULL, ‘2001-05-21’, ‘Friendly Fred’, 152000, 7)
(‘Gobi’, NULL, ‘2001-05-21’, NULL, 2348000, 100)
(‘Gobi’, NULL, NULL, NULL, 1339800000, 58000)
(NULL, NULL, NULL, NULL, 3521727000, 198000)
```

Recall that NULL is used to indicate a rolled-up dimension, equivalent to the \* we used in the abstract CUBE operator’s result. □

A variant of the CUBE operator, called ROLLUP, produces the additional aggregated tuples only if they aggregate over a tail of the sequence of grouping attributes. We indicate this option by appending WITH ROLLUP to the group-by clause.

**Example 10.33:** We can get the part of the data cube for Sales that is constructed by the ROLLUP operator with:

```
CREATE MATERIALIZED VIEW SalesRollup AS
    SELECT model, color, date, dealer, SUM(val), SUM(cnt)
    FROM Sales
    GROUP BY model, color, date, dealer WITH ROLLUP;
```

The view SalesRollup will contain tuples

```
(‘Gobi’, ‘red’, ‘2001-05-21’, ‘Friendly Fred’, 45000, 2)
(‘Gobi’, ‘red’, ‘2001-05-21’, NULL, 3678000, 135)
(‘Gobi’, ‘red’, NULL, NULL, 657100000, 34566)
(‘Gobi’, NULL, NULL, NULL, 1339800000, 58000)
(NULL, NULL, NULL, NULL, 3521727000, 198000)
```

because these tuples represent aggregation along some dimension and all dimensions, if any, that follow it in the list of grouping attributes.

However, SalesRollup would not contain tuples such as

```
('Gobi', NULL, '2001-05-21', 'Friendly Fred', 152000, 7)
('Gobi', NULL, '2001-05-21', NULL, 2348000, 100)
```

These each have `NULL` in a dimension (`color` in both cases) but do not have `NULL` in one or more of the following dimension attributes. □

### 10.7.3 Exercises for Section 10.7

**Exercise 10.7.1:** What is the ratio of the size of  $\text{CUBE}(F)$  to the size of  $F$  if fact table  $F$  has the following characteristics?

- a)  $F$  has ten dimension attributes, each with ten different values.
- b)  $F$  has ten dimension attributes, each with two different values.

**Exercise 10.7.2:** Use the materialized view `SalesCube` from Example 10.32 to answer the following queries:

- a) Find the total sales of blue cars for each dealer.
- b) Find the total number of green Gobis sold by dealer "Smilin' Sally."
- c) Find the average number of Gobis sold on each day of March, 2007 by each dealer.

! **Exercise 10.7.3:** What help, if any, would the rollup `SalesRollup` of Example 10.33 be for each of the queries of Exercise 10.7.2?

**Exercise 10.7.4:** In Exercise 10.6.1 we spoke of PC-order data organized as a fact table with dimension tables for attributes `cust`, `proc`, `memory`, `hd`, and `od`. That is, each tuple of the fact table `Orders` has an ID for each of these attributes, leading to information about the PC involved in the order. Write a SQL query that will produce the data cube for this fact table.

**Exercise 10.7.5:** Answer the following queries using the data cube from Exercise 10.7.4. If necessary, use dimension tables as well. You may invent suitable names and attributes for the dimension tables.

- a) Find, for each processor speed, the total number of computers ordered in each month of the year 2007.
- b) List for each type of hard disk (e.g., SCSI or IDE) and each processor type the number of computers ordered.
- c) Find the average price of computers with 3.0 gigahertz processors for each month from Jan., 2005.

! **Exercise 10.7.6:** The cube tuples mentioned in Example 10.32 are not in the rollup of Example 10.33. Are there other rollups that would contain these tuples?

**!! Exercise 10.7.7:** If the fact table  $F$  to which we apply the CUBE operator is sparse (i.e., there are many fewer tuples in  $F$  than the product of the number of possible values along each dimension), then the ratio of the sizes of  $\text{CUBE}(F)$  and  $F$  can be very large. How large can it be?

## 10.8 Summary of Chapter 10

- ◆ *Privileges*: For security purposes, SQL systems allow many different kinds of privileges to be managed for database elements. These privileges include the right to select (read), insert, delete, or update relations, the right to reference relations (refer to them in a constraint), and the right to create triggers.
- ◆ *Grant Diagrams*: Privileges may be granted by owners to other users or to the general user PUBLIC. If granted with the grant option, then these privileges may be passed on to others. Privileges may also be revoked. The grant diagram is a useful way to remember enough about the history of grants and revocations to keep track of who has what privilege and from whom they obtained those privileges.
- ◆ *SQL Recursive Queries*: In SQL, one can define a relation recursively — that is, in terms of itself. Or, several relations can be defined to be mutually recursive.
- ◆ *Monotonicity*: Negations and aggregations involved in a SQL recursion must be monotone — inserting tuples in one relation does not cause tuples to be deleted from any relation, including itself. Intuitively, a relation may not be defined, directly or indirectly, in terms of a negation or aggregation of itself.
- ◆ *The Object-Relational Model*: An alternative to pure object-oriented database models like ODL is to extend the relational model to include the major features of object-orientation. These extensions include nested relations, i.e., complex types for attributes of a relation, including relations as types. Other extensions include methods defined for these types, and the ability of one tuple to refer to another through a reference type.
- ◆ *User-Defined Types in SQL*: Object-relational capabilities of SQL are centered around the UDT, or user-defined type. These types may be declared by listing their attributes and other information, as in table declarations. In addition, methods may be declared for UDT's.
- ◆ *Relations With a UDT as Type*: Instead of declaring the attributes of a relation, we may declare that relation to have a UDT. If we do so, then its tuples have one component, and this component is an object of the UDT.

- ◆ *Reference Types:* A type of an attribute can be a reference to a UDT. Such attributes essentially are pointers to objects of that UDT.
- ◆ *Object Identity for UDT's:* When we create a relation whose type is a UDT, we declare an attribute to serve as the “object-ID” of each tuple. This component is a reference to the tuple itself. Unlike in object-oriented systems, this “OID” column may be accessed by the user, although it is rarely meaningful.
- ◆ *Accessing components of a UDT:* SQL provides observer and mutator functions for each attribute of a UDT. These functions, respectively, return and change the value of that attribute when applied to any object of that UDT.
- ◆ *Ordering Functions for UDT's:* In order to compare objects, or to use SQL operations such as DISTINCT, GROUP BY, or ORDER BY, it is necessary for the implementer of a UDT to provide a function that tells whether two objects are equal or whether one precedes the other.
- ◆ *OLAP:* On-line analytic processing involves complex queries that touch all or much of the data, at the same time. Often, a separate database, called a data warehouse, is constructed to run such queries while the actual database is used for short-term transactions (OLTP, or on-line transaction processing).
- ◆ *ROLAP and MOLAP:* It is frequently useful, for OLAP queries, to think of the data as residing in a multidimensional space, with dimensions corresponding to independent aspects of the data represented. Systems that support such a view of data take either a relational point of view (ROLAP, or relational-OLAP systems), or use the specialized data-cube model (MOLAP, or multidimensional-OLAP systems).
- ◆ *Star Schemas:* In a star schema, each data element (e.g., a sale of an item) is represented in one relation, called the fact table, while information helping to interpret the values along each dimension (e.g., what kind of product is item 1234?) is stored in a dimension table for each dimension.
- ◆ *The Cube Operator:* A specialized operator called cube pre-aggregates the fact table along all subsets of dimensions. It may add little to the space needed by the fact table, and greatly increases the speed with which many OLAP queries can be answered.
- ◆ *Data Cubes in SQL:* We can turn the result of a query into a data cube by appending WITH CUBE to a group-by clause. We can also construct a portion of the cube by using WITH ROLLUP there.

## 10.9 References for Chapter 10

The ideas behind the SQL authorization mechanism originated in [4] and [1].

Material on object-relational features of SQL can be obtained as described in the bibliographic notes to Chapter 6.

The source of the SQL-99 proposal for recursion is [2]. This proposal, and its monotonicity requirement, built on foundations developed over many years, involving recursion and negation in Datalog; see [5].

The cube operator was proposed in [3].

1. R. Fagin, “On an authorization mechanism,” *ACM Transactions on Database Systems* 3:3, pp. 310–319, 1978.
2. S. J. Finkelstein, N. Mattos, I. S. Mumick, and H. Pirahesh, “Expressing recursive queries in SQL,” ISO WG3 report X3H2-96-075, March, 1996.
3. J. N. Gray, A. Bosworth, A. Layman, and H. Pirahesh, “Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals,” *Proc. Intl. Conf. on Data Engineering* (1996), pp. 152–159.
4. P. P. Griffiths and B. W. Wade, “An authorization mechanism for a relational database system,” *ACM Transactions on Database Systems* 1:3, pp. 242–255, 1976.
5. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.

# **Part III**

# **Modeling and Programming for Semistructured Data**



# Chapter 11

# The Semistructured-Data Model

We now turn to a different kind of data model. This model, called “semistructured,” is distinguished by the fact that the schema is implied by the data, rather than being declared separately from the data as is the case for the relational model and all the other models we studied up to this point. After a general discussion of semistructured data, we turn to the most important manifestation of this idea: XML. We shall cover ways to describe XML data, in effect enforcing a schema for this “schemaless” data. These methods include DTD’s (Document Type Definitions) and the language XML Schema.

## 11.1 Semistructured Data

The *semistructured-data* model plays a special role in database systems:

1. It serves as a model suitable for *integration* of databases, that is, for describing the data contained in two or more databases that contain similar data with different schemas.
2. It serves as the underlying model for notations such as XML, to be taken up in Section 11.2, that are being used to share information on the Web.

In this section, we shall introduce the basic ideas behind “semistructured data” and how it can represent information more flexibly than the other models we have met previously.

### 11.1.1 Motivation for the Semistructured-Data Model

The models we have seen so far — E/R, UML, relational, ODL — each start with a schema. The schema is a rigid framework into which data is placed. This

rigidity provides certain advantages. Especially, the relational model owes much of its success to the existence of efficient implementations. This efficiency comes from the fact that the data in a relational database must fit the schema, and the schema is known to the query processor. For instance, fixing the schema allows the data to be organized with data structures that support efficient answering of queries, as we discussed in Section 8.3.

On the other hand, interest in the semistructured-data model is motivated primarily by its flexibility. In particular, semistructured data is “schemaless.” More precisely, the data is *self-describing*; it carries information about what its schema is, and that schema can vary arbitrarily, both over time and within a single database.

One might naturally wonder whether there is an advantage to creating a database without a schema, where one could enter data at will, and attach to the data whatever schema information you felt was appropriate for that data. There are actually some small-scale information systems, such as Lotus Notes, that take the self-describing-data approach. This flexibility may make query processing harder, but it offers significant advantages to users. For example, we can maintain a database of movies in the semistructured model and add new attributes like “would I like to see this movie?” as we wish. The attributes do not need to have a value for all movies, or even for more than one movie. Likewise, we can add relationships like “homage to,” without having to change the schema or even represent the relationship in more than one pair of movies.

### 11.1.2 Semistructured Data Representation

A database of *semistructured data* is a collection of *nodes*. Each node is either a *leaf* or *interior*. Leaf nodes have associated data; the type of this data can be any atomic type, such as numbers and strings. Interior nodes have one or more arcs out. Each arc has a *label*, which indicates how the node at the head of the arc relates to the node at the tail. One interior node, called the *root*, has no arcs entering and represents the entire database. Every node must be reachable from the root, although the graph structure is not necessarily a tree.

**Example 11.1:** Figure 11.1 is an example of a semistructured database about stars and movies. We see a node at the top labeled *Root*; this node is the entry point to the data and may be thought of as representing all the information in the database. The central objects or entities — stars and movies in this case — are represented by nodes that are children of the root.

We also see many leaf nodes. At the far left is a leaf labeled *Carrie Fisher*, and at the far right is a leaf labeled *1977*, for instance. There are also many interior nodes. Three particular nodes we have labeled *cf*, *mh*, and *sw*, standing for “Carrie Fisher,” “Mark Hamill,” and “Star Wars,” respectively. These labels are not part of the model, and we placed them on these nodes only so we would have a way of referring to the nodes, which otherwise would be nameless, in the text. We may think of node *sw*, for instance, as representing the concept “Star

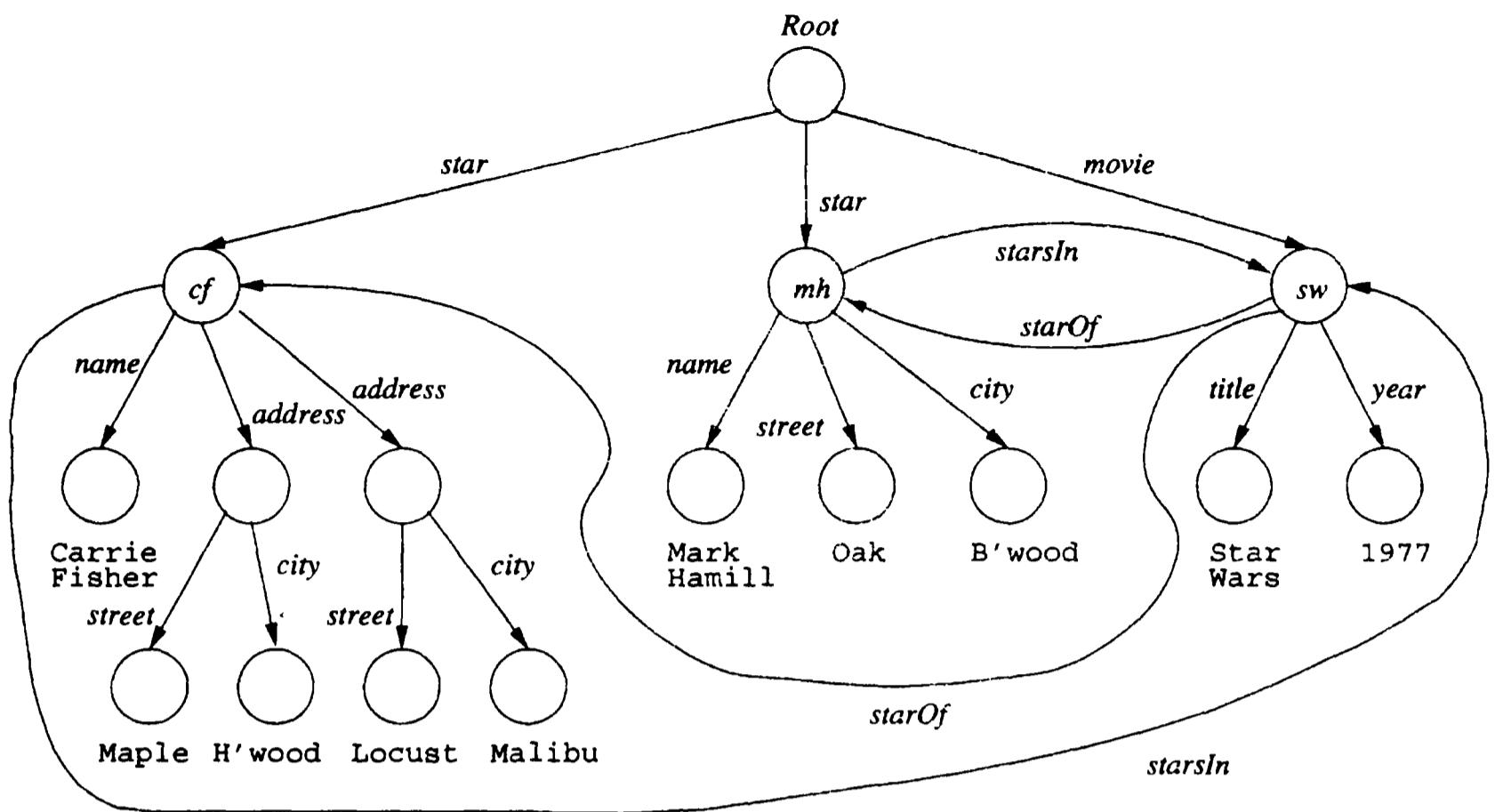


Figure 11.1: Semistructured data representing a movie and stars

Wars": the title and year of this movie, other information not shown, such as its length, and its stars, two of which are shown. □

A label  $L$  on the arc from node  $N$  to node  $M$  can play one of two roles:

1. It may be possible to think of  $N$  as representing an object or entity, while  $M$  represents one of its attributes. Then,  $L$  represents the name of the attribute.
2. We may be able to think of  $N$  and  $M$  as objects or entities and  $L$  as the name of a relationship from  $N$  to  $M$ .

**Example 11.2:** Consider Fig. 11.1 again. The node indicated by  $cf$  may be thought of as representing the **Star** object for Carrie Fisher. We see, leaving this node, an arc labeled **name**, which represents the attribute **name** and leads to a leaf node holding the correct name. We also see two arcs, each labeled **address**. These arcs lead to unnamed nodes which we may think of as representing two addresses of Carrie Fisher. There is no schema to tell us whether stars can have more than one address; we simply put two address nodes in the graph if we feel it is appropriate.

Notice in Fig. 11.1 how both nodes have out-arcs labeled **street** and **city**. Moreover, these arcs each lead to leaf nodes with the appropriate atomic values. We may think of **address** nodes as structs or objects with two fields, named **street** and **city**. However, in the semistructured model, it is entirely appropriate to add other components, e.g., **zip**, to some addresses, or to have one or both fields missing.

The other kind of arc also appears in Fig. 11.1. For instance, the node *cf* has an out-arc leading to the node *sw* and labeled *starsIn*. The node *mh* (for Mark Hamill) has a similar arc, and the node *sw* has arcs labeled *starOf* to both nodes *cf* and *mh*. These arcs represent the stars-in relationship between stars and movies. □

### 11.1.3 Information Integration Via Semistructured Data

The flexibility and self-describing nature of semistructured data has made it important in two applications. We shall discuss its use for data exchange in Section 11.2, but here we shall consider its use as a tool for information integration. As databases have proliferated, it has become a common requirement that data in two or more of them be accessible as if they were one database. For instance, companies may merge; each has its own personnel database, its own database of sales, inventory, product designs, and perhaps many other matters. If corresponding databases had the same schemas, then combining them would be simple; for instance, we could take the union of the tuples in two relations that had the same schema and played the same roles in the the two databases.

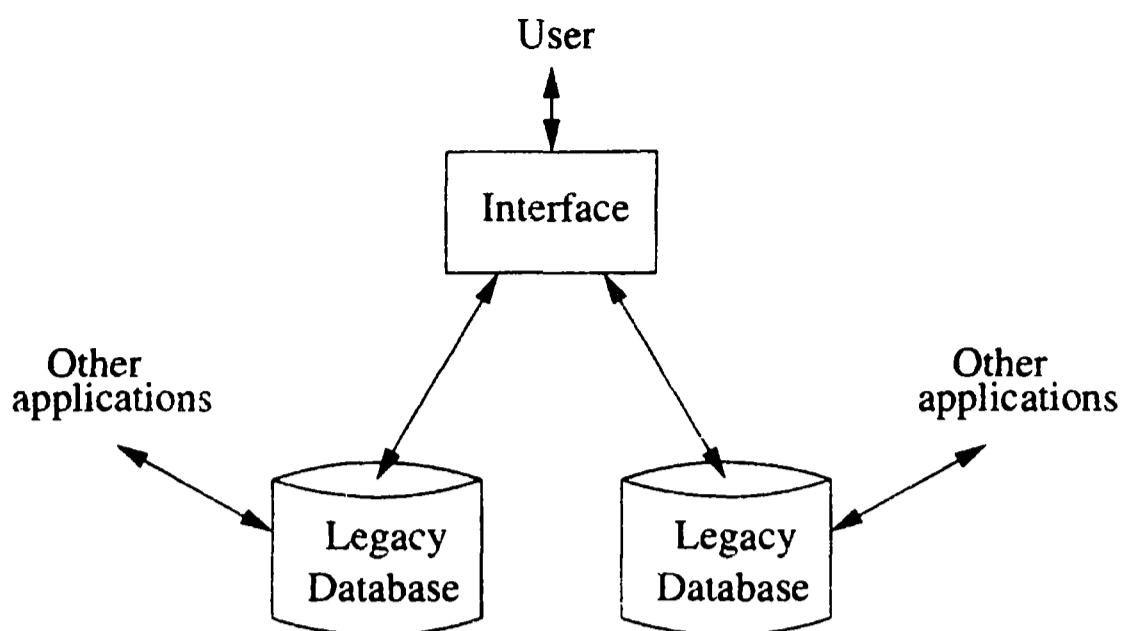
However, life is rarely that simple. Independently developed databases are unlikely to share a schema, even if they talk about the same things, such as personnel. For instance, one employee database may record spouse-name, another not. One may have a way to represent several addresses, phones, or emails for an employee, another database may allow only one of each. One may treat consultants as employees, another not. One database might be relational, another object-oriented.

To make matters more complex, databases tend over time to be used in so many different applications that it is impossible to turn them off and copy or translate their data into another database, even if we could figure out an efficient way to transform the data from one schema to another. This situation is often referred to as the *legacy-database problem*; once a database has been in existence for a while, it becomes impossible to disentangle it from the applications that grow up around it, so the database can never be decommissioned.

A possible solution to the legacy-database problem is suggested in Fig. 11.2. We show two legacy databases with an interface; there could be many legacy systems involved. The legacy systems are each unchanged, so they can support their usual applications.

For flexibility in integration, the interface supports semistructured data, and the user is allowed to query the interface using a query language that is suitable for such data. The semistructured data may be constructed by translating the data at the sources, using components called *wrappers* (or “adapters”) that are each designed for the purpose of translating one source to semistructured data.

Alternatively, the semistructured data at the interface may not exist at all. Rather, the user queries the interface as if there were semistructured data, while the interface answers the query by posing queries to the sources, each referring to the schema found at that source.



**Figure 11.2:** Integrating two legacy databases through an interface that supports semistructured data

**Example 11.3:** We can see in Fig. 11.1 a possible effect of information about stars being gathered from several sources. Notice that the address information for Carrie Fisher has an address concept, and the address is then broken into street and city. That situation corresponds roughly to data that had a nested-relation schema like `Stars(name, address(street, city))`.

On the other hand, the address information for Mark Hamill has no address concept at all, just street and city. This information may have come from a schema such as `Stars(name, street, city)` that can represent only one address for a star. Some of the other variations in schema that are not reflected in the tiny example of Fig. 11.1, but that could be present if movie information were obtained from several sources, include: optional film-type information, a director, a producer or producers, the owning studio, revenue, and information on where the movie is currently playing. □

#### 11.1.4 Exercises for Section 11.1

**Exercise 11.1.1:** Since there is no schema to design in the semistructured-data model, we cannot ask you to design schemas to describe different situations. Rather, in the following exercises we shall ask you to suggest how particular data might be organized to reflect certain facts.

- Add to Fig. 11.1 the facts that *Star Wars* was directed by George Lucas and produced by Gary Kurtz.
- Add to Fig. 11.1 information about *Empire Strikes Back* and *Return of the Jedi*, including the facts that Carrie Fisher and Mark Hamill appeared in these movies.
- Add to (b) information about the studio (Fox) for these movies and the address of the studio (Hollywood).

**Exercise 11.1.2:** Suggest how typical data about banks and customers, as in Exercise 4.1.1, could be represented in the semistructured model.

**Exercise 11.1.3:** Suggest how typical data about players, teams, and fans, as was described in Exercise 4.1.3, could be represented in the semistructured model.

**Exercise 11.1.4:** Suggest how typical data about a genealogy, as was described in Exercise 4.1.6, could be represented in the semistructured model.

! **Exercise 11.1.5:** UML and the semistructured-data model are both “graphical” in nature, in the sense that they use nodes, labels, and connections among nodes as the medium of expression. Yet there is an essential difference between the two models. What is it?

## 11.2 XML

XML (*Extensible Markup Language*) is a tag-based notation designed originally for “marking” documents, much like the familiar HTML. Nowadays, data with XML “markup” can be represented in many ways. However, in this section we shall refer to XML data as represented in one or more documents. While HTML’s tags talk about the presentation of the information contained in documents — for instance, which portion is to be displayed in italics or what the entries of a list are — XML tags are intended to talk about the meanings of pieces of the document.

In this section we shall introduce the rudiments of XML. We shall see that it captures, in a linear form, the same structure as do the graphs of semistructured data introduced in Section 11.1. In particular, tags can play the same role as the labels on the arcs of a semistructured-data graph.

### 11.2.1 Semantic Tags

Tags in XML are text surrounded by triangular brackets, i.e., `<...>`, as in HTML. Also as in HTML, tags generally come in matching pairs, with an *opening tag* like `<Foo>` and a matched *closing tag* that is the same word with a slash, like `</Foo>`. Between a matching pair `<Foo>` and `</Foo>`, there can be text, including text with nested HTML tags, and any number of other nested matching pairs of XML tags. A pair of matching tags and everything that comes between them is called an *element*.

A single tag, with no matched closing tag, is also permitted in XML. In this form, the tag has a slash before the right bracket, for example, `<Foo/>`. Such a tag cannot have any other elements or text nested within it. It can, however, have attributes (see Section 11.2.4).

### 11.2.2 XML With and Without a Schema

XML is designed to be used in two somewhat different modes:

1. *Well-formed* XML allows you to invent your own tags, much like the arc-labels in semistructured data. This mode corresponds quite closely to semistructured data, in that there is no predefined schema, and each document is free to use whatever tags the author of the document wishes. Of course the nesting rule for tags must be obeyed, or the document is not well-formed.
2. *Valid* XML involves a “DTD,” or “Document Type Definition” (see Section 11.3) that specifies the allowable tags and gives a grammar for how they may be nested. This form of XML is intermediate between the strict-schema models such as the relational model, and the completely schemaless world of semistructured data. As we shall see in Section 11.3, DTD’s generally allow more flexibility in the data than does a conventional schema; DTD’s often allow optional fields or missing fields, for instance.

### 11.2.3 Well-Formed XML

The minimal requirement for well-formed XML is that the document begin with a declaration that it is XML, and that it have a *root element* that is the entire body of the text. Thus, a well-formed XML document would have an outer structure like:

```
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<SomeTag>
  ...
</SomeTag>
```

The first line indicates that the file is an XML document. The encoding UTF-8 (UTF = “Unicode Transformation Format”) is a common choice of encoding for characters in documents, because it is compatible with ASCII and uses only one byte for the ASCII characters. The attribute `standalone = "yes"` indicates that there is no DTD for this document; i.e., it is well-formed XML. Notice that this initial declaration is delineated by special markers `<?...?>`. The root element for this document is labeled `<SomeTag>`.

**Example 11.4:** In Fig. 11.3 is an XML document that corresponds roughly to the data in Fig. 11.1. In particular, it corresponds to the tree-like portion of the semistructured data — the root and all the nodes and arcs except the “sideways” arcs among the nodes *cf*, *mh*, and *sw*. We shall see in Section 11.2.4 how those may be represented.

The root element is `StarMovieData`. Within this element, we see two elements, each beginning with the tag `<Star>` and ending with its matching

```

<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
  <Star>
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maple St.</Street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ln.</Street>
      <City>Malibu</City>
    </Address>
  </Star>
  <Star>
    <Name>Mark Hamill</Name>
    <Street>456 Oak Rd.</Street>
    <City>Brentwood</City>
  </Star>
  <Movie>
    <Title>Star Wars</Title>
    <Year>1977</Year>
  </Movie>
</StarMovieData>

```

Figure 11.3: An XML document about stars and movies

</Star>. Within each element is a subelement giving the name of the star. One element, for Carrie Fisher, also has two subelements, each giving the address of one of her homes. These elements are each delineated by an <Address> opening tag and its matched closing tag. The element for Mark Hamill has only subelements for one street and one city, and does not use an <Address> tag to group these. This distinction appeared as well in Fig. 11.1. We also see one element with opening tag <Movie> and its matched closing tag. This element has subelements for the title and year of the movie.

Notice that the document of Fig. 11.3 does not represent the relationship “stars-in” between stars and movies. We could indicate the movies of a star by including, within the element devoted to that star, the titles and years of their movies. Figure 11.4 is an example of this representation. □

#### 11.2.4 Attributes

As in HTML, an XML element can have *attributes* (name-value pairs) within its opening tag. An attribute is an alternative way to represent a leaf node of semistructured data. Attributes, like tags, can represent labeled arcs in

```

<Star>
  <Name>Mark Hamill</Name>
  <Street>Oak</Street>
  <City>Brentwood</City>
  <Movie>
    <Title>Star Wars</Title>
    <Year>1977</Year>
  </Movie>
  <Movie>
    <Title>Empire Strikes Back</Title>
    <Year>1980</Year>
  </Movie>
</Star>

```

Figure 11.4: Nesting movies within stars

a semistructured-data graph. Attributes can also be used to represent the “sideways” arcs as in Fig. 11.1.

**Example 11.5:** The *title* or *year* children of the movie node labeled *sw* could be represented directly in the `<Movie>` element, rather than being represented by nested elements. That is, we could replace the `<Movie>` element of Fig. 11.3 by:

```
<Movie year = 1977><Title>Star Wars</Title></Movie>
```

We could even make both child nodes be attributes by:

```
<Movie title = "Star Wars" year = 1977></Movie>
```

or even:

```
<Movie title = "Star Wars" year = 1977 />
```

Notice that here we use a single tag without a matched closing tag, as indicated by the slash at the end. □

### 11.2.5 Attributes That Connect Elements

An important use for attributes is to represent connections in a semistructured data graph that do not form a tree. We shall see in Section 11.3.4 how to declare certain attributes to be identifiers for their elements. We shall also see how to declare that other attributes are references to these element identifiers. For the moment, let us just see an example of how these attributes could be used.

**Example 11.6:** Figure 11.5 can be interpreted as an exact representation in XML of the semistructured data graph of Fig. 11.1. However, in order to make the interpretation, we need to have enough schema information that we know the attribute `starID` is an identifier for the element in which it appears. That is, `cf` is the identifier of the first `<Star>` element (for Carrie Fisher) and `mh` is the identifier of the second `<Star>` element (for Mark Hamill). Likewise, we must establish that the attribute `movieID` within a `<Movie>` tag is an identifier for that element. Thus, `sw` is an identifier for the lone `<Movie>` element in Fig. 11.5.

```

<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
    <Star starID = "cf" starredIn = "sw">
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        </Address>
    </Star>
    <Star starID = "mh" starredIn = "sw">
        <Name>Mark Hamill</Name>
        <Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Star>
    <Movie movieID = "sw" starsOf = "cf", "mh">
        <Title>Star Wars</Title>
        <Year>1977</Year>
    </Movie>
</StarMovieData>

```

Figure 11.5: Adding stars-in information to our XML document

Moreover, the schema must also say that the attributes `starredIn` for `<Star>` elements and `starsOf` for `<Movie>` elements are references to one or more ID's. That is, the value `sw` for `starredIn` within each of the `<Movie>` elements says that both Carrie Fisher and Mark Hamill starred in *Star Wars*. Likewise, the list of ID's `cf` and `mh` that is the value of `starsOf` in the `<Movie>` element says that both these stars were stars of *Star Wars*. □

### 11.2.6 Namespaces

There are situations in which XML data involves tags that come from two or more different sources, and which may therefore have conflicting names. For example, we would not want to confuse an HTML tag used in text with an XML tag that represents the meaning of that text. In Section 11.4, we shall see how XML Schema requires tags from two separate vocabularies. To distinguish among different vocabularies for tags in the same document, we can use a *namespace* for a set of tags.

To say that an element's tag should be interpreted as part of a certain namespace, we can use the attribute `xmlns` in its opening tag. There is a special form used for this attribute:

$$\text{xmlns:}name = "URI"$$

Within the element having this attribute, *name* can modify any tag to say the tag belongs to this namespace. That is, we can create *qualified* names of the form *name:tag*, where *name* is the name of the namespace to which the tag *tag* belongs.

The URI (Universal Resource Identifier) is typically a URL referring to a document that describes the meaning of the tags in the namespace. This description need not be formal; it could be an informal article about expectations. It could even be nothing at all, and still serve the purpose of distinguishing different tags that had the same name.

**Example 11.7:** Suppose we want to say that in element `StarMovieData` of Fig. 11.5 certain tags belong to the namespace defined in the document `infolab.stanford.edu/movies`. We could choose a name such as `md` for the namespace by using the opening tag:

```
<md:StarMovieData xmlns:md=
    "http://infolab.stanford.edu/movies">
```

Our intent is that `StarMovieData` itself is part of this namespace, so it gets the prefix `md:`, as does its closing tag `/md:StarMovieData`. Inside this element, we have the option of asserting that the tags of subelements belong to this namespace by prefixing their opening and closing tags with `md:`. □

### 11.2.7 XML and Databases

Information encoded in XML is not always intended to be stored in a database. It has become common for computers to share data across the Internet by passing messages in the form of XML elements. These messages live for a very short time, although they may have been generated using data from one database and wind up being stored as tuples of a database at the receiving end. For example, the XML data in Fig. 11.5 might be turned into some tuples

to insert into relations `MovieStar` and `StarsIn` of our running example movie database.

However, it is becoming increasingly common for XML to appear in roles traditionally reserved for relational databases. For example, we discussed in Section 11.1.3 how systems that integrate the data of an enterprise produce integrated views of many databases. XML is becoming an important option as the way to represent these views, as an alternative to views consisting of relations or classes of objects. The integrated views are then queried using one of the specialized XML query languages that we shall meet in Chapter 12.

When we store XML in a database, we must deal with the requirement that access to information must be efficient, especially for very large XML documents or very large collections of small documents.<sup>1</sup> A relational DBMS provides indexes and other tools for making access efficient, a subject we introduced in Section 8.3. There are two approaches to storing XML to provide some efficiency:

1. Store the XML data in a parsed form, and provide a library of tools to navigate the data in that form. Two common standards are called SAX (Simple API for XML) and DOM (Document Object Model).
2. Represent the documents and their elements as relations, and use a conventional, relational DBMS to store them.

In order to represent XML documents as relations, we should start by giving each document and each element of those documents a unique ID. For the document, the ID could be its URL or path in a file system. A possible relational database schema is:

```
DocRoot(docID, rootElementID)
SubElement(parentID, childID, position)
ElementAttribute(elementID, name, value)
ElementValue(elementID, value)
```

This schema is suitable for documents that obey the restriction that each element either contains only text or contains only subelements. Accommodating elements with *mixed content* of text and subelements is left as an exercise.

The first relation, `DocRoot` relates document ID's to the ID's of their root element. The second relation, `SubElement`, connects an element (the “parent”) to each of its immediate subelements (“children”). The third attribute of `SubElement` gives the position of the child among all the children of the parent.

The third relation, `ElementAttribute` relates elements to their attributes; each tuple gives the name and value of one of the attributes of an element. Finally, `ElementValue` relates those elements that have no subelements to the text, if any, that is contained in that element.

---

<sup>1</sup>Recall that XML data need not take the form of documents (i.e., a header with a root element) at all. For example, XML data could be a stream of elements without headers. However, we shall continue to speak of “documents” as XML data.

There is a small matter that values of attributes and elements can have different types, e.g., integers or strings, while relational attributes each have a unique type. We could treat the two attributes named `value` as always being strings, and interpret those strings that were integers or another type properly as we processed the data. Or we could split each of the last two relations into as many relations as there are different types of data.

### 11.2.8 Exercises for Section 11.2

**Exercise 11.2.1:** Repeat Exercise 11.1.1 using XML.

**Exercise 11.2.2:** Show that any relation can be represented by an XML document. *Hint:* Create an element for each tuple with a subelement for each component of that tuple.

! **Exercise 11.2.3:** How would you represent an empty element (one that had neither text nor subelements) in the database schema of Section 11.2.7?

! **Exercise 11.2.4:** In Section 11.2.7 we gave a database schema for representing documents that do not have *mixed content* — elements that contain a mixture of text (#PCDATA) and subelements. Show how to modify the schema when elements can have mixed content.

## 11.3 Document Type Definitions

For a computer to process XML documents automatically, it is helpful for there to be something like a schema for the documents. It is useful to know what kinds of elements can appear in a collection of documents and how elements can be nested. The description of the schema is given by a grammar-like set of rules, called a *document type definition*, or DTD. It is intended that companies or communities wishing to share data will each create a DTD that describes the form(s) of the data they share, thus establishing a shared view of the semantics of their elements. For instance, there could be a DTD for describing protein structures, a DTD for describing the purchase and sale of auto parts, and so on.

### 11.3.1 The Form of a DTD

The gross structure of a DTD is:

```
<!DOCTYPE root-tag [
    <!ELEMENT element-name (components)>
        more elements
]>
```

The opening *root-tag* and its matched closing tag surround a document that conforms to the rules of this DTD. Element declarations, introduced by **!ELEMENT**, give the tag used to surround the portion of the document that represents the element, and also give a parenthesized list of “components.” The latter are elements that may or must appear in the element being described. The exact requirements on components are indicated in a manner we shall see shortly.

There are two important special cases of components:

1. (#PCDATA) (“parsed character data”) after an element name means that element has a value that is text, and it has no elements nested within. Parsed character data may be thought of as HTML text. It can have formatting information within it, and the special characters like < must be escaped, by &lt;; and similar HTML codes. For instance,

```
<!ELEMENT Title (#PCDATA)>
```

says that between **<Title>** and **</Title>** tags a character string can appear. However, any nested tags are not part of the XML; they could be HTML, for instance.

2. The keyword **EMPTY**, with no parentheses, indicates that the element is one of those that has no matched closing tag. It has no subelements, nor does it have text as a value. For instance,

```
<!ELEMENT Foo EMPTY>
```

says that the only way the tag **Foo** can appear is as **<Foo/>**.

**Example 11.8:** In Fig. 11.6 we see a DTD for stars.<sup>2</sup> The DTD name and root element is **Stars**. The first element definition says that inside the matching pair of tags **<Stars>...</Stars>** we shall find zero or more **Star** elements, each representing a single star. It is the \* in **(Star\*)** that says “zero or more,” i.e., “any number of.”

The second element, **Star**, is declared to consist of three kinds of subelements: **Name**, **Address**, and **Movies**. They must appear in this order, and each must be present. However, the + following **Address** says “one or more”; that is, there can be any number of addresses listed for a star, but there must be at least one. The **Name** element is then defined to be parsed character data. The fourth element says that an address element consists of subelements for a street and a city, in that order.

Then, the **Movies** element is defined to have zero or more elements of type **Movie** within it; again, the \* says “any number of.” A **Movie** element is defined to consist of title and year elements, each of which are simple text. Figure 11.7 is an example of a document that conforms to the DTD of Fig. 11.6. □

```
<!DOCTYPE Stars [
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star (Name, Address+, Movies)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (Street, City)>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

Figure 11.6: A DTD for movie stars

The components of an element  $E$  are generally other elements. They must appear between the tags  $\langle E \rangle$  and  $\langle /E \rangle$  in the order listed. However, there are several operators that control the number of times elements appear.

1. A \* following an element means that the element may occur any number of times, including zero times.
2. A + following an element means that the element may occur one or more times.
3. A ? following an element means that the element may occur either zero times or one time, but no more.
4. We can connect a list of options by the “or” symbol | to indicate that exactly one option appears. For example, if  $\langle \text{Movie} \rangle$  elements had  $\langle \text{Genre} \rangle$  subelements, we might declare these by

```
<!ELEMENT Genre (Comedy|Drama|SciFi|Teen)>
```

to indicate that each  $\langle \text{Genre} \rangle$  element has one of these four subelements.

5. Parentheses can be used to group components. For example, if we declared addresses to have the form

```
<!ELEMENT Address Street, (City|Zip)>
```

then  $\langle \text{Address} \rangle$  elements would each have a  $\langle \text{Street} \rangle$  subelement followed by either a  $\langle \text{City} \rangle$  or  $\langle \text{Zip} \rangle$  subelement, but not both.

---

<sup>2</sup>Note that the stars-and-movies XML document of Fig. 11.3 is not intended to conform to this DTD.

```

<Stars>
  <Star>
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maple St.</Street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ln.</Street>
      <City>Malibu</City>
    </Address>
    <Movies>
      <Movie>
        <Title>Star Wars</Title>
        <Year>1977</Year>
      </Movie>
      <Movie>
        <Title>Empire Strikes Back</Title>
        <Year>1980</Year>
      </Movie>
      <Movie>
        <Title>Return of the Jedi</Title>
        <Year>1983</Year>
      </Movie>
    </Movies>
  </Star>
  <Star>
    <Name>Mark Hamill</Name>
    <Address>
      <Street>456 Oak Rd.<Street>
      <City>Brentwood</City>
    </Address>
    <Movies>
      <Movie>
        <Title>Star Wars</Title>
        <Year>1977</Year>
      </Movie>
      <Movie>
        <Title>Empire Strikes Back</Title>
        <Year>1980</Year>
      </Movie>
      <Movie>
        <Title>Return of the Jedi</Title>
        <Year>1983</Year>
      </Movie>
    </Movies>
  </Star>
</Stars>

```

Figure 11.7: Example of a document following the DTD of Fig. 11.6

### 11.3.2 Using a DTD

If a document is intended to conform to a certain DTD, we can either:

- a) Include the DTD itself as a preamble to the document, or
- b) In the opening line, refer to the DTD, which must be stored separately in the file system accessible to the application that is processing the document.

**Example 11.9:** Here is how we might introduce the document of Fig. 11.7 to assert that it is intended to conform to the DTD of Fig. 11.6.

```
<?xml version = "1.0" encoding = "utf-8" standalone = "no"?>
<!DOCTYPE Stars SYSTEM "star.dtd">
```

The attribute `standalone = "no"` says that a DTD is being used. Recall we set this attribute to `"yes"` when we did not wish to specify a DTD for the document. The location from which the DTD can be obtained is given in the `!DOCTYPE` clause, where the keyword `SYSTEM` followed by a file name gives this location. □

### 11.3.3 Attribute Lists

A DTD also lets us specify which attributes an element may have, and what the types of these attributes are. A declaration of the form

```
<!ATTLIST element-name attribute-name type >
```

says that the named attribute can be an attribute of the named element, and that the type of this attribute is the indicated type. Several attributes can be defined in one `ATTLIST` statement, but it is not necessary to do so, and the `ATTLIST` statements can appear in any position in the DTD.

The most common type for attributes is `CDATA`. This type is essentially character-string data with special characters like `<` escaped as in `#PCDATA`. Notice that `CDATA` does not take a pound sign as `#PCDATA` does. Another option is an enumerated type, which is a list of possible strings, surrounded by parentheses and separated by `|`'s. Following the data type there can be a keyword `#REQUIRED` or `#IMPLIED`, which means that the attribute must be present, or is optional, respectively.

**Example 11.10:** Instead of having the title and year be subelements of a `<Movie>` element, we could make these be attributes instead. Figure 11.8 shows possible attribute-list declarations. Notice that `Movie` is now an empty element. We have given it three attributes: `title`, `year`, and `genre`. The first two are `CDATA`, while the `genre` has values from an enumerated type. Note that in the document, the values, such as `comedy`, appear with quotes. Thus,

```
<Movie title = "Star Wars" year = "1977" genre = "sciFi" />
```

is a possible movie element in a document that conforms to this DTD. □

```
<!ELEMENT Movie EMPTY>
  <!ATTLIST Movie
    title CDATA #REQUIRED
    year CDATA #REQUIRED
    genre (comedy | drama | sciFi | teen) #IMPLIED
  >
```

Figure 11.8: Data about movies will appear as attributes

```
<!DOCTYPE StarMovieData [
  <!ELEMENT StarMovieData (Star*, Movie*)>
  <!ELEMENT Star (Name, Address+)>
    <!ATTLIST Star
      starId ID #REQUIRED
      starredIn IDREFS #IMPLIED
    >
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Address (Street, City)>
  <!ELEMENT Street (#PCDATA)>
  <!ELEMENT City (#PCDATA)>
  <!ELEMENT Movie (Title, Year)>
    <!ATTLIST Movie
      movieId ID #REQUIRED
      starsOf IDREFS #IMPLIED
    >
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Year (#PCDATA)>
]>
```

Figure 11.9: A DTD for stars and movies, using ID's and IDREF's

### 11.3.4 Identifiers and References

Recall from Section 11.2.5 that certain attributes can be used as identifiers for elements. In a DTD, we give these attributes the type ID. Other attributes have values that are references to these element ID's; these attributes may be declared to have type IDREF. The value of an IDREF attribute must also be the value of some ID attribute of some element, so the IDREF is in effect a pointer to the ID. An alternative is to give an attribute the type IDREFS. In that case, the value of the attribute is a string consisting of a list of ID's, separated by whitespace. The effect is that an IDREFS attribute links its element to a set of elements — the elements identified by the ID's on the list.

**Example 11.11:** Figure 11.9 shows a DTD in which stars and movies are given equal status, and the ID-IDREFS correspondence is used to describe the many-many relationship between movies and stars that was suggested in the semistructured data of Fig. 11.1. The structure differs from that of the DTD in Fig. 11.6, in that stars and movies have equal status; both are subelements of the root element. That is, the name of the root element for this DTD is **StarMovieData**, and its elements are a sequence of stars followed by a sequence of movies.

A star no longer has a set of movies as subelements, as was the case for the DTD of Fig. 11.6. Rather, its only subelements are a name and address, and in the beginning **<Star>** tag we shall find an attribute **starredIn** of type **IDREFS**, whose value is a list of ID's for the movies of the star.

```
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
    <Star starID = "cf" starredIn = "sw">
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        </Address>
    </Star>
    <Star starID = "mh" starredIn = "sw">
        <Name>Mark Hamill</Name>
        <Address>
            <Street>456 Oak Rd.</Street>
            <City>Brentwood</City>
        </Address>
    </Star>
    <Movie movieID = "sw" starsOf = "cf mh">
        <Title>Star Wars</Title>
        <Year>1977</Year>
    </Movie>
</StarMovieData>
```

Figure 11.10: Adding stars-in information to our XML document

A **<Star>** element also has an attribute **starId**. Since it is declared to be of type **ID**, the value of **starId** may be referenced by **<Movie>** elements to indicate the stars of the movie. That is, when we look at the attribute list for **Movie** in Fig. 11.9, we see that it has an attribute **movieId** of type **ID**; these

are the ID's that will appear on lists that are the values of `starredIn` elements. Symmetrically, the attribute `starsOf` of `Movie` is an IDREFS, a list of ID's for stars. □

### 11.3.5 Exercises for Section 11.3

**Exercise 11.3.1:** Add to the document of Fig. 11.10 the following facts:

- a) Carrie Fisher and Mark Hamill also starred in *The Empire Strikes Back* (1980) and *Return of the Jedi* (1983).
- b) Harrison Ford also starred in *Star Wars*, in the two movies mentioned in (a), and the movie *Firewall* (2006).
- c) Carrie Fisher also starred in *Hannah and Her Sisters* (1985).
- d) Matt Damon starred in *The Bourne Identity* (2002).

**Exercise 11.3.2:** Suggest how typical data about banks and customers, as was described in Exercise 4.1.1, could be represented as a DTD.

**Exercise 11.3.3:** Suggest how typical data about players, teams, and fans, as was described in Exercise 4.1.3, could be represented as a DTD.

**Exercise 11.3.4:** Suggest how typical data about a genealogy, as was described in Exercise 4.1.6, could be represented as a DTD.

**! Exercise 11.3.5:** Using your representation from Exercise 11.2.2, devise an algorithm that will take any relation schema (a relation name and a list of attribute names) and produce a DTD describing a document that represents that relation.

## 11.4 XML Schema

*XML Schema* is an alternative way to provide a schema for XML documents. It is more powerful than DTD's, giving the schema designer extra capabilities. For instance, XML Schema allows arbitrary restrictions on the number of occurrences of subelements. It allows us to declare types, such as integer or float, for simple elements, and it gives us the ability to declare keys and foreign keys.

### 11.4.1 The Form of an XML Schema

An XML Schema description of a schema is itself an XML document. It uses the namespace at the URL:

<http://www.w3.org/2001/XMLSchema>

that is provided by the World-Wide-Web Consortium. Each XML-Schema document thus has the form:

```
<? xml version = "1.0" encoding = "utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    ...
</xs:schema>
```

The first line indicates XML, and uses the special brackets <? and ?>. The second line is the root tag for the document that is the schema. The attribute `xmlns` (XML namespace) makes the variable `xs` stand for the namespace for XML Schema that was mentioned above. It is this namespace that causes the tag `<xs:schema>` to be interpreted as `schema` in the namespace for XML Schema. As discussed in Section 11.2.6, qualifying each XML-Schema term we use with the prefix `xs:` will cause each such tag to be interpreted according to the rules for XML Schema. Between the opening `<xs:schema>` tag and its matched closing tag `</xs:schema>` will appear a schema. In what follows, we shall learn the most important tags from the XML-Schema namespace and what they mean.

## 11.4.2 Elements

An important component of schemas is the *element*, which is similar to an element definition in a DTD. In the discussion that follows, you should be alert to the fact that, because XML-Schema definitions are XML documents, these schemas are themselves composed of “elements.” However, the elements of the schema itself, each of which has a tag that begins with `xs:`, are not the elements being defined by the schema.<sup>3</sup> The form of an element definition in XML Schema is:

```
<xs:element name = element name type = element type >
    constraints and/or structure information
</xs:element>
```

The element name is the chosen tag for these elements in the schema being defined. The type can be either a simple type or a complex type. Simple types include the common primitive types, such as `xs:integer`, `xs:string`, and `xs:boolean`. There can be no subelements for an element of a simple type.

**Example 11.12:** Here are title and year elements defined in XML Schema:

```
<xs:element name = "Title" type = "xs:string" />
<xs:element name = "Year" type = "xs:integer" />
```

---

<sup>3</sup>To further assist in the distinction between tags that are part of a schema definition and the tags of the schema being defined, we shall begin each of the latter with a capital letter.

Each of these `<xs:element>` elements is itself empty, so it can be closed by `/>` with no matched closing tag. The first defined element has name `Title` and is of string type. The second element is named `Year` and is of type integer. In documents (perhaps talking about movies) with `<Title>` and `<Year>` elements, these elements will not be empty, but rather will be followed by a string (the title) or integer (the year), and a matched closing tag, `</Title>` or `</Year>`, respectively. □

### 11.4.3 Complex Types

A *complex type* in XML Schema can have several forms, but the most common is a sequence of elements. These elements are required to occur in the sequence given, but the number of repetitions of each element can be controlled by attributes `minOccurs` and `maxOccurs`, that appear in the element definitions themselves. The meanings of these attributes are as expected; no fewer than `minOccurs` occurrences of each element may appear in the sequence, and no more than `maxOccurs` occurrences may appear. If there is more than one occurrence, they must all appear consecutively. The default, if one or both of these attributes are missing, is one occurrence. To say that there is no upper limit on occurrences, use the value "unbounded" for `maxOccurs`.

```

<xs:complexType name = type namelist of element definitions
  </xs:sequence>
</xs:complexType>

```

Figure 11.11: Defining a complex type that is a sequence of elements

The form of a definition for a complex-type that is a sequence of elements is shown in Fig. 11.11. The name for the complex type is optional, but is needed if we are going to use this complex type as the type of one or more elements of the schema being defined. An alternative is to place the complex-type definition between an opening `<xs:element>` tag and its matched closing tag, to make that complex type be the type of the element.

**Example 11.13:** Let us write a complete XML-Schema document that defines a very simple schema for movies. The root element for movie documents will be `<Movies>`, and the root will have zero or more `<Movie>` subelements. Each `<Movie>` element will have two subelements: a title and year, in that order. The XML-Schema document is shown in Fig. 11.12.

Lines (1) and (2) are a typical preamble to an XML-Schema definition. In lines (3) through (8), we define a complex type whose name is `movieType`. This type consists of a sequence of two elements named `Title` and `Year`; they are the elements we saw in Example 11.12. The type definition itself does not

```

1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">

3)      <xs:complexType name = "movieType">
4)          <xs:sequence>
5)              <xs:element name = "Title" type = "xs:string" />
6)              <xs:element name = "Year" type = "xs:integer" />
7)          </xs:sequence>
8)      </xs:complexType>

9)      <xs:element name = "Movies">
10)         <xs:complexType>
11)             <xs:sequence>
12)                 <xs:element name = "Movie" type = "movieType"
13)                     minOccurs = "0" maxOccurs = "unbounded" />
14)             </xs:sequence>
15)         </xs:complexType>
16)     </xs:element>

17) </xs:schema>

```

Figure 11.12: A schema for movies in XML Schema

create any elements, but notice how the name `movieType` is used in line (12) to make this type be the type of `Movie` elements.

Lines (9) through (15) define the element `Movies`. Although we could have created a complex type for this element, as we did for `Movie`, we have chosen to include the type in the element definition itself. Thus, we put no type attribute in line (9). Rather, between the opening `<xs:element>` tag at line (9) and its matched closing tag at line (15) appears a complex-type definition for the element `Movies`. This complex type has no name, but it is defined at line (11) to be a sequence. In this case, the sequence has only one kind of element, `Movie`, as indicated by line (12). This element is defined to have type `movieType` — the complex type we defined at lines (3) through (8). It is also defined to have between zero and infinity occurrences. Thus, the schema of Fig. 11.12 says the same thing as the DTD we show in Fig. 11.13. □

There are several other ways we can construct a complex type.

- In place of `xs:sequence` we could use `xs:all`, which means that each of the elements between the opening `<xs:all>` tag and its matched closing tag must occur, in any order, exactly once each.
- Alternatively, we could replace `xs:sequence` by `xs:choice`. Then, exactly one of the elements found between the opening `<xs:choice>` tag

```
<!DOCTYPE Movies [
    <!ELEMENT Movies (Movie*)
    <!ELEMENT Movie (Title, Year)
    <!ELEMENT Title (#PCDATA)
    <!ELEMENT Year (#PCDATA)
]>
```

Figure 11.13: A DTD for movies

and its matched closing tag will appear.

The elements inside a sequence or choice can have `minOccurs` and `maxOccurs` attributes to govern how many times they can appear. In the case of a choice, only one of the elements can appear at all, but it can appear more than once if it has a value of `maxOccurs` greater than 1. The rules for `xs:all` are different. It is not permitted to have a `maxOccurs` value other than 1, but `minOccurs` can be either 0 or 1. In the former case, the element might not appear at all.

#### 11.4.4 Attributes

A complex type can have attributes. That is, when we define a complex type  $T$ , we can include instances of element `<xs:attribute>`. When we use  $T$  as the type of an element  $E$ , then  $E$  can have (or must have) an instance of this attribute. The form of an attribute definition is:

```
<xs:attribute name = attribute name type = type name
    other information about the attribute />
```

The “other information” may include information such as a default value and usage (required or optional — the latter is the default).

**Example 11.14:** The notation

```
<xs:attribute name = "year" type = "xs:integer"
    default = "0" />
```

defines `year` to be an attribute of type `integer`. We do not know of what element `year` is an attribute; it depends where the above definition is placed. The default value of `year` is 0, meaning that if an element without a value for attribute `year` occurs in a document, then the value of `year` is taken to be 0.

As another instance:

```
<xs:attribute name = "year" type = "xs:integer"
    use = "required" />
```

is another definition of the attribute `year`. However, setting `use` to `required` means that any element of the type being defined must have a value for attribute `year`. □

Attribute definitions are placed within a complex-type definition. In the next example, we rework Example 11.13 by making the type `movieType` have attributes for the title and year, rather than subelements for that information.

```

1) <? xml version = "1.0" encoding = "utf-8" ?>
2) <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">

3)   <xs:complexType name = "movieType">
4)     <xs:attribute name = "title" type = "xs:string"
      use = "required" />
5)     <xs:attribute name = "year" type = "xs:integer"
      use = "required" />
6)   </xs:complexType>

7)   <xs:element name = "Movies">
8)     <xs:complexType>
9)       <xs:sequence>
10)         <xs:element name = "Movie" type = "movieType"
           minOccurs = "0" maxOccurs = "unbounded" />
11)       </xs:sequence>
12)     </xs:complexType>
13)   </xs:element>

14) </xs:schema>
```

Figure 11.14: Using attributes in place of simple elements

**Example 11.15:** Figure 11.14 shows the revised XML Schema definition. At lines (4) and (5), the attributes `title` and `year` are defined to be required attributes for elements of type `movieType`. When element `Movie` is given that type at line (10), we know that every `<Movie>` element must have values for `title` and `year`. Figure 11.15 shows the DTD resembling Fig. 11.14. □

## 11.4.5 Restricted Simple Types

It is possible to create a restricted version of a simple type such as `integer` or `string` by limiting the values the type can take. These types can then be used as the type of an attribute or element. We shall consider two kinds of restrictions here:

```
<!DOCTYPE Movies [
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie EMPTY>
        <!ATTLIST Movie
            title CDATA #REQUIRED
            year  CDATA #REQUIRED
        >
    ]>
```

Figure 11.15: DTD equivalent for Fig. 11.14

1. Restricting numerical values by using `minInclusive` to state the lower bound, `maxInclusive` to state the upper bound.<sup>4</sup>
2. Restricting values to an enumerated type.

The form of a range restriction is shown in Fig. 11.16. The restriction has a base, which may be a primitive type (e.g., `xs:string`) or another simple type.

```
<xs:simpleType name = type name >
    <xs:restriction base = base type >
        upper and/or lower bounds
    </xs:restriction>
</xs:simpleType>
```

Figure 11.16: Form of a range restriction

**Example 11.16:** Suppose we want to restrict the year of a movie to be no earlier than 1915. Instead of using `xs:integer` as the type for element `Year` in line (6) of Fig. 11.12 or for the attribute `year` in line (5) of Fig. 11.14, we could define a new simple type as in Fig. 11.17. The type `movieYearType` would then be used in place of `xs:integer` in the two lines cited above. □

Our second way to restrict a simple type is to provide an enumeration of values. The form of a single enumerated value is:

```
<xs:enumeration value = some value />
```

A restriction can consist of any number of these values.

---

<sup>4</sup>The “inclusive” means that the range of values includes the given bound. An alternative is to replace `Inclusive` by `Exclusive`, meaning that the stated bounds are just outside the permitted range.

```

<xs:simpleType name = "movieYearType">
    <xs:restriction base = "xs:integer">
        <xs:minInclusive value = "1915" />
    </xs:restriction>
<xs:simpleType>

```

Figure 11.17: A type that restricts integer values to be 1915 or greater

**Example 11.17:** Let us design a simple type suitable for the genre of movies. In our running example, we have supposed that there are only four possible genres: comedy, drama, sciFi, and teen. Figure 11.18 shows how to define a type `genreType` that could serve as the type for an element or attribute representing our genres of movies. □

```

<xs:simpleType name = "genreType">
    <xs:restriction base = "xs:string">
        <xs:enumeration value = "comedy" />
        <xs:enumeration value = "drama" />
        <xs:enumeration value = "sciFi" />
        <xs:enumeration value = "teen" />
    </xs:restriction>
<xs:simpleType>

```

Figure 11.18: A enumerated type in XML Schema

## 11.4.6 Keys in XML Schema

An element can have a key declaration, which says that when we look at a certain class  $C$  of elements, values of one or more given *fields* within those elements are unique. The concept of “field” is actually quite general, but the most common case is for a field to be either a subelement or an attribute. The class  $C$  of elements is defined by a “selector.” Like fields, selectors can be complex, but the most common case is a sequence of one or more element names, each a subelement of the one before it. In terms of a tree of semistructured data, the class is all those nodes reachable from a given node by following a particular sequence of arc labels.

**Example 11.18:** Suppose we want to say, about the semistructured data in Fig. 11.1, that among all the nodes we can reach from the root by following a *star* label, what we find following a further *name* label leads us to a unique value. Then the “selector” would be *star* and the “field” would be *name*. The implication of asserting this key is that within the root element shown, there

cannot be two stars with the same name. If movies had names instead of titles, then the key assertion would not prevent a movie and a star from having the same name. Moreover, if there were actually many elements like the tree of Fig. 11.1 found in one document (e.g., each of the objects we called “Root” in that figure were actually a single movie and its stars), then different trees could have the same star name without violating the key constraint. □

The form of a key declaration is

```
<xs:key name = key name >
  <xs:selector xpath = path description >
    <xs:field xpath = path description >
  </xs:key>
```

There can be more than one line with an **xs:field** element, in case several fields are needed to form the key. An alternative is to use the element **xs:unique** in place of **xs:key**. The difference is that if “key” is used, then the fields must exist for each element defined by the selector. However, if “unique” is used, then they might not exist, and the constraint is only that they are unique if they exist.

The selector path can be any sequence of elements, each of which is a subelement of the previous. The element names are separated by slashes. The field can be any subelement of the last element on the selector path, or it can be an attribute of that element. If it is an attribute, then it is preceded by the “at-sign.” There are other options, and in fact, the selector and field can be any XPath expressions; we take up the XPath query language in Section 12.1.

**Example 11.19:** In Fig. 11.19 we see an elaboration of Fig. 11.12. We have added the element **Genre** to the definition of **movieType**, in order to have a nonkey subelement for a movie. Lines (3) through (10) define **genreType** as in Example 11.17. The **Genre** subelement of **movieType** is added at line (15).

The definition of the **Movies** element has been changed in lines (24) through (28) by the addition of a key. The name of the key is **movieKey**; this name will be used if it is referenced by a foreign key, as we shall discuss in Section 11.4.7. Otherwise, the name is irrelevant. The selector path is just **Movie**, and there are two fields, **Title** and **Year**. The meaning of this key declaration is that, within any **Movies** element, among all its **Movie** subelements, no two can have both the same title and the same year, nor can any of these values be missing. Note that because of the way **movieType** was defined at lines (13) and (14), with no values for **minOccurs** or **maxOccurs** for **Title** or **Year**, the defaults, 1, apply, and there must be exactly one occurrence of each. □

#### 11.4.7 Foreign Keys in XML Schema

We can also declare that an element has, perhaps deeply nested within it, a field or fields that serve as a reference to the key for some other element. This

```
1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">

3)  <xs:simpleType name = "genreType">
4)      <xs:restriction base = "xs:string">
5)          <xs:enumeration value = "comedy" />
6)          <xs:enumeration value = "drama" />
7)          <xs:enumeration value = "sciFi" />
8)          . <xs:enumeration value = "teen" />
9)      </xs:restriction>
10) <xs:simpleType>

11)     <xs:complexType name = "movieType">
12)         <xs:sequence>
13)             <xs:element name = "Title" type = "xs:string" />
14)             <xs:element name = "Year" type = "xs:integer" />
15)             <xs:element name = "Genre" type = "genreType"
16)                 minOccurs = "0" maxOccurs = "1" />
17)         </xs:sequence>
18)     </xs:complexType>
19)     <xs:element name = "Movies">
20)         <xs:complexType>
21)             <xs:sequence>
22)                 <xs:element name = "Movie" type = "movieType"
23)                     minOccurs = "0" maxOccurs = "unbounded" />
24)             </xs:sequence>
25)         </xs:complexType>
26)         <xs:key name = "movieKey">
27)             <xs:selector xpath = "Movie" />
28)             <xs:field xpath = "Title" />
29)             <xs:field xpath = "Year" />
30)         </xs:key>
31)     </xs:element>

32) </xs:schema>
```

Figure 11.19: A schema for movies in XML Schema

capability is similar to what we get with ID's and IDREF's in a DTD (see Section 11.3.4). However, the latter are untyped references, while references in XML Schema are to particular types of elements. The form of a foreign-key definition in XML Schema is:

```
<xs:keyref name = foreign-key name refer = key name >
  <xs:selector xpath = path description >
    <xs:field xpath = path description >
  </xs:keyref>
```

The schema element is `xs:keyref`. The foreign-key itself has a name, and it refers to the name of some key or unique value. The selector and field(s) are as for keys.

**Example 11.20:** Figure 11.20 shows the definition of an element `<Stars>`. We have used the style of XML Schema where each complex type is defined within the element that uses it. Thus, we see at lines (4) through (6) that a `<Stars>` element consists of one or more `<Star>` subelements.

At lines (7) through (11), we see that each `<Star>` element has three kinds of subelements. There is exactly one `<Name>` and one `<Address>` subelement, and any number of `<StarredIn>` subelements. In lines (12) through (15), we find that a `<StarredIn>` element has no subelements, but it does have two attributes, `title` and `year`.

Lines (22) through (26) define a foreign key. In line (22) we see that the name of this foreign-key constraint is `movieRef` and that it refers to the key `movieKey` that was defined in Fig. 11.19. Notice that this foreign key is defined within the `<Stars>` definition. The selector is `Star/StarredIn`. That is, it says we should look at every `<StarredIn>` subelement of every `<Star>` subelement of a `<Stars>` element. From that `<StarredIn>` element, we extract the two fields `title` and `year`. The `@` indicates that these are attributes rather than subelements. The assertion made by this foreign-key constraint is that any title-year pair we find in this way will appear in some `<Movie>` element as the pair of values for its subelements `<Title>` and `<Year>`. □

#### 11.4.8 Exercises for Section 11.4

**Exercise 11.4.1:** Give an example of a document that conforms to the XML Schema definition of Fig. 11.12 and an example of one that has all the elements mentioned, but does not conform to the definition.

**Exercise 11.4.2:** Rewrite Fig. 11.12 so that there is a named complex type for `Movies`, but no named type for `Movie`.

**Exercise 11.4.3:** Write the XML Schema definitions of Fig. 11.19 and 11.20 as a DTD.

```
1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
3)    <xs:element name = "Stars">
4)      <xs:complexType>
5)        <xs:sequence>
6)          <xs:element name = "Star" minOccurs = "1"
7)                      maxOccurs = "unbounded">
8)          <xs:complexType>
9)            <xs:sequence>
10)              <xs:element name = "Name"
11)                type = "xs:string" />
12)              <xs:element name = "Address"
13)                type = "xs:string" />
14)              <xs:element name = "StarredIn"
15)                minOccurs = "0"
16)                maxOccurs = "unbounded">
17)                  <xs:complexType>
18)                    <xs:attribute name = "title"
19)                      type = "xs:string" />
20)                    <xs:attribute name = "year"
21)                      type = "xs:integer" />
22)                  </xs:complexType>
23)                </xs:element>
24)              </xs:sequence>
25)            </xs:complexType>
26)          </xs:element>
27)        </xs:sequence>
28)      </xs:complexType>
29)    <xs:keyref name = "movieRef" refers = "movieKey">
30)      <xs:selector xpath = "Star/StarredIn" />
31)      <xs:field xpath = "@title" />
32)      <xs:field xpath = "@year" />
33)    </xs:keyref>
34)  </xs:element>
```

Figure 11.20: Stars with a foreign key

## 11.5 Summary of Chapter 11

- ◆ *Semistructured Data*: In this model, data is represented by a graph. Nodes are like objects or values of attributes, and labeled arcs connect an object to both the values of its attributes and to other objects to which it is connected by a relationship.
- ◆ *XML*: The Extensible Markup Language is a World-Wide-Web Consortium standard that represents semistructured data linearly.
- ◆ *XML Elements*: Elements consist of an opening tag `<Foo>`, a matched closing tag `</Foo>`, and everything between them. What appears can be text, or it can be subelements, nested to any depth.
- ◆ *XML Attributes*: Tags can have attribute-value pairs within them. These attributes provide additional information about the element with which they are associated.
- ◆ *Document Type Definitions*: The DTD is a simple, grammatical form of defining elements and attributes of XML, thus providing a rudimentary schema for those XML documents that use the DTD. An element is defined to have a sequence of subelements, and these elements can be required to appear exactly once, at most once, at least once, or any number of times. An element can also be defined to have a list of required and/or optional attributes.
- ◆ *Identifiers and References in DTD's*: To represent graphs that are not trees, a DTD allows us to declare attributes of type ID and IDREF(S). An element can thus be given an identifier, and that identifier can be referred to by other elements from which we would like to establish a link.
- ◆ *XML Schema*: This notation is another way to define a schema for certain XML documents. XML Schema definitions are themselves written in XML, using a set of tags in a namespace that is provided by the World-Wide-Web Consortium.
- ◆ *Simple Types in XML Schema*: The usual sorts of primitive types, such as integers and strings, are provided. Additional simple types can be defined by restricting a simple type, such as by providing a range for values or by giving an enumeration of permitted values.
- ◆ *Complex Types in XML Schema*: Structured types for elements may be defined to be sequences of elements, each with a minimum and maximum number of occurrences. Attributes of an element may also be defined in its complex type.
- ◆ *Keys and Foreign Keys in XML Schema*: A set of elements and/or attributes may be defined to have a unique value within the scope of some

enclosing element. Other sets of elements and/or attributes may be defined to have a value that appears as a key within some other kind of element.

## 11.6 References for Chapter 11

Semistructured data as a data model was first studied in [5] and [4]. LOREL, the prototypical query language for this model is described in [3]. Surveys of work on semistructured data include [1], [7], and the book [2].

XML is a standard developed by the World-Wide-Web Consortium. The home page for information about XML is [9]. References on DTD's and XML Schema are also found there. For XML parsers, the definition of DOM is in [8] and for SAX it is [6]. A useful place to go for quick tutorials on many of these subjects is [10].

1. S. Abiteboul, “Querying semi-structured data,” *Proc. Intl. Conf. on Database Theory* (1997), Lecture Notes in Computer Science 1187 (F. Afrati and P. Kolaitis, eds.), Springer-Verlag, Berlin, pp. 1–18.
2. S. Abiteboul, D. Suciu, and P. Buneman, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan-Kaufmann, San Francisco, 1999.
3. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Weiner, “The LOREL query language for semistructured data,” In *J. Digital Libraries* 1:1, 1997.
4. P. Buneman, S. B. Davidson, and D. Suciu, “Programming constructs for unstructured data,” *Proceedings of the Fifth International Workshop on Database Programming Languages*, Gubbio, Italy, Sept., 1995.
5. 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.
6. Sax Project, <http://www.saxproject.org/>
7. D. Suciu (ed.) Special issue on management of semistructured data, *SIGMOD Record* 26:4 (1997).
8. World-Wide-Web Consortium, <http://www.w3.org/DOM/>
9. World-Wide-Web Consortium, <http://www.w3.org/XML/>
10. W3 Schools, <http://www.w3schools.com>



# Chapter 12

# Programming Languages for XML

We now turn to programming languages for semistructured data. All the widely used languages of this type apply to XML data, and might be used for semistructured data represented in other ways as well. In this chapter, we shall study three such languages. The first, XPath, is a simple language for describing sets of similar paths in a graph of semistructured data. XQuery is an extension of XPath that adopts something of the style of SQL. It allows iterations over sets, subqueries, and many other features that will be familiar from the study of SQL.

The third topic of this chapter is XSLT. This language was developed originally as a transformation language, capable of restructuring XML documents or turning them into printable (HTML) documents. However, its expressive power is actually quite similar to that of XQuery, and it is capable of producing XML results. Thus, it can serve as a query language for XML.

## 12.1 XPath

In this section, we introduce XPath. We begin with a discussion of the data model used in the most recent version of XPath, called XPath 2.0; this model is used in XQuery as well. This model plays a role analogous to the “bag of tuples of primitive-type components” that is used in the relational model as the value of a relation.

In later sections, we learn about XPath path expressions and their meaning. In general, these expressions allow us to move from elements of a document to some or all of their subelements. Using “axes,” we are also able to move within documents in a variety of ways, and to obtain the attributes of elements.

### 12.1.1 The XPath Data Model

As in the relational model, XPath assumes that all values — those it produces and those constructed in intermediate steps — have the same general “shape.” In the relational model, this “shape” is a bag of tuples. Tuples in a given bag all have the same number of components, and the components each have a primitive type, e.g., integer or string. In XPath, the analogous “shape” is *sequence of items*. An *item* is either:

1. A value of primitive type: integer, real, boolean, or string, for example.
2. A *node*. There are many kinds of nodes, but in our introduction, we shall only talk about three kinds:
  - (a) *Documents*. These are files containing an XML document, perhaps denoted by their local path name or a URL.
  - (b) *Elements*. These are XML elements, including their opening tags, their matched closing tag if there is one, and everything in between (i.e., below them in the tree of semistructured data that an XML document represents).
  - (c) *Attributes*. These are found inside opening tags, as we discussed in several places in Chapter 11.

The items in a sequence need not be all of the same type, although often they will be.

**Example 12.1:** Figure 12.1 is a sequence of four items. The first is the integer 10; the second is a string, and the third is a real. These are all items of primitive type.

```

10
"ten"
10.0

<Number base = "8">
  <Digit>1</Digit>
  <Digit>2</Digit>
</Number>

@val="10"

```

Figure 12.1: A sequence of five items

The fourth item is a node, and this node’s type is “element.” Notice that the element has tag **Number** with an attribute and two subelements with tag **Digit**. The last item is an attribute node. □

### 12.1.2 Document Nodes

While the documents to which XPath is applied can come from various sources, it is common to apply XPath to documents that are files. We can make a document node from a file by applying the function:

```
doc(file name)
```

The named file should be an XML document. We can name a file either by giving its local name or a URL if it is remote. Thus, examples of document nodes include:

```
doc("movies.xml")
doc("/usr/sally/data/movies.xml")
doc("infolab.stanford.edu/~hector/movies.xml")
```

Every XPath query refers to a document. In many cases, this document will be apparent from the context. For example, recall our discussion of XML-Schema keys in Section 11.4.6. We used XPath expressions to denote the selector and field(s) for a key. In that context, the document was “whatever document the schema definition is being applied to.”

### 12.1.3 Path Expressions

Typically, an XPath expression starts at the root of a document and gives a sequence of tags and slashes (/), say  $/T_1/T_2/\cdots/T_n$ . We evaluate this expression by starting with a sequence of items consisting of one node: the document. We then process each of  $T_1, T_2, \dots$  in turn. To process  $T_i$ , consider the sequence of items that results from processing the previous tags, if any. Examine those items, in order, and find for each all its subelements whose tag is  $T_i$ . Those items are appended to the output sequence, in the order in which they appear in the document.

As a special case, the root tag  $T_1$  for the document is considered a “subelement” of the document node. Thus, the expression  $/T_1$  produces a sequence of one item, which is an element node consisting of the entire contents of the document. The difference may appear subtle; before we applied the expression  $/T_1$ , we had a document node representing the file, and after applying  $/T_1$  to that node we have an element node representing the text in the file.

**Example 12.2:** Suppose our document is a file containing the XML text of Fig. 11.5, which we reproduce here as Fig. 12.2. The path expression  $/StarMovieData$  produces the sequence of one element. This element has tag `<StarMovieData>`, of course, and it consists of everything in Fig. 12.2 except for line (1).

Now, consider the path expression

```
/StarMovieData/Star/Name
```

```

1)  <? xml version="1.0" encoding="utf-8" standalone="yes" ?>
2)  <StarMovieData>
3)      <Star starID = "cf" starredIn = "sw">
4)          <Name>Carrie Fisher</Name>
5)          <Address>
6)              <Street>123 Maple St.</Street>
7)              <City>Hollywood</City>
8)          </Address>
9)          <Address>
10)             <Street>5 Locust Ln.</Street>
11)             <City>Malibu</City>
12)         </Address>
13)     </Star>
14)     <Star starID = "mh" starredIn = "sw">
15)         <Name>Mark Hamill</Name>
16)         <Street>456 Oak Rd.</Street>
17)         <City>Brentwood</City>
18)     </Star>
19)     <Movie movieID = "sw" starsOf = "cf", "mh">
20)         <Title>Star Wars</Title>
21)         <Year>1977</Year>
22)     </Movie>
23) </StarMovieData>

```

Figure 12.2: An XML document for applying path expressions

When we apply the **StarMovieData** tag to the sequence consisting of the document, we get the sequence consisting of the root element, as discussed above. Next, we apply to this sequence the tag **Star**. There are two subelements of the **StarMovieData** element that have tag **Star**. These are lines (3) through (12) for star Carrie Fisher and lines (14) through (18) for star Mark Hamill. Thus, the result of the path expression **/StarMovieData/Star** is the sequence of these two elements, in that order.

Finally, we apply to this sequence the tag **Name**. The first element has one **Name** subelement, at line (4). The second element also has one **Name** subelement, at line (15). Thus, the sequence

```

<Name>Carrie Fisher</Name>
<Name>Mark Hamill</Name>

```

is the result of applying the path expression **/StarMovieData/Star/Name** to the document of Fig. 12.2. □

### 12.1.4 Relative Path Expressions

In several contexts, we shall use XPath expressions that are *relative* to the current node or sequence of nodes.

- In Section 11.4.6 we talked about selector and field values that were really XPath expressions relative to a node or sequence of nodes for which we were defining a key.
- In Example 12.2 we talked about applying the XPath expression **Star** to the element consisting of the entire document, or the expression **Name** to a sequence of **Star** elements.

Relative expressions do not start with a slash. Each such expression must be applied in some context, which will be clear from its use. The similarity to the way files and directories are designated in a UNIX file system is not accidental.

### 12.1.5 Attributes in Path Expressions

Path expressions allow us to find all the elements within a document that are reached from the root along a particular kind of path (a sequence of tags). Sometimes, we want to find not these elements but rather the values of an attribute of those elements. If so, we can end the path expression by an attribute name preceded by an at-sign. That is, the path-expression form is  $/T_1/T_2/\dots/T_n/@A$ .

The result of this expression is computed by first applying the path expression  $/T_1/T_2/\dots/T_n$  to get a sequence of elements. We then look at the opening tag of each element, in turn, to find an attribute  $A$ . If there is one, then the value of that attribute is appended to the sequence that forms the result.

**Example 12.3:** The path expression

`/StarMovieData/Star/@starID`

applied to the document of Fig. 12.2 finds the two **Star** elements and looks into their opening tags at lines (3) and (14) to find the values of their **starID** attributes. Both elements have this attribute, so the result sequence is "cf" "mh".

□

### 12.1.6 Axes

So far, we have only navigated through semistructured-data graphs in two ways: from a node to its children or to an attribute. XPath in fact provides a large number of *axes*, which are modes of navigation. Two of these axes are *child* (the default axis) and *attribute*, for which `@` is really a shorthand. At each step in a path expression, we can prefix a tag or attribute name by an axis name and a double-colon. For example,

`/StarMovieData/Star/@starID`

is really shorthand for:

`/child::StarMovieData/child::Star/attribute::starID`

Some of the other axes are parent, ancestor (really a proper ancestor), descendant (a proper descendant), next-sibling (any sibling to the right), previous-sibling (any sibling to the left), self, and descendant-or-self. The latter has a shorthand `//` and takes us from a sequence of elements to those elements and all their subelements, at any level of nesting.

**Example 12.4:** It might look hard to find, in the document of Fig. 12.2, all the cities where stars live. The problem is that Mark Hamill’s city is not nested within an `Address` element, so it is not reached along the same paths as Carrie Fisher’s cities. However, the path expression

`//City`

finds all the `City` subelements, at any level of nesting, and returns them in the order in which they appear in the document. That is, the result of this path expression is the sequence:

```
<City>Hollywood</City>
<City>Malibu</City>
<City>Brentwood</City>
```

which we obtain from lines (7), (11), and (17), respectively.

We could also use the `//` axis within the path expression. For example, should the document contain city information that wasn’t about stars (e.g., studios and their addresses), then we could restrict the paths that we consider to make sure that the city was a subelement of a `Star` element. For the given document, the path expression

`/StarMovieData/Star//City`

produces the same three `City` elements as a result. □

Some of the other axes have shorthands as well. For example, `..` stands for parent, and `.` for self. We have already seen `@` for attribute and `/` for child.

### 12.1.7 Context of Expressions

In order to understand the meaning of an axis like parent, we need to explore further the view of data in XPath. Results of expressions are sequences of elements or primitive values. However, XPath expressions and their results do not exist in isolation; if they did, it would not make sense to ask for the “parent” of an element. Rather, there is normally a context in which the expression is

evaluated. In all our examples, there is a single document from which elements are extracted. If we think of an element in the result of some XPath expression as a reference to the element in the document, then it makes sense to apply axes like parent, ancestor, or next-sibling to the element in the sequence.

For example, we mentioned in Section 11.4.6 that keys in XML Schema are defined by a pair of XPath expressions. Key constraints apply to XML documents that obey the schema that includes the constraint. Each such document provides the context for the XPath expressions in the schema itself. Thus, it is permitted to use all the XPath axes in these expressions.

### 12.1.8 Wildcards

Instead of specifying a tag along every step of a path, we can use a `*` to say “any tag.” Likewise, instead of specifying an attribute, `@*` says “any attribute.”

**Example 12.5:** Consider the path expression

```
/StarMovieData/*/@*
```

applied to the document of Fig. 12.2. First, `/StarMovieData/*` takes us to every subelement of the root element. There are three: two stars and a movie. Thus, the result of this path expression is the sequence of elements in lines (3) through (13), (14) through (18), and (19) through (22).

However, the expression asks for the values of all the attributes of these elements. We therefore look for attributes among the outermost tags of each of these elements, and return their values in the order in which they appear in the document. Thus, the sequence

```
"cf" "sw" "mh" "sw" "sw" "cf" "mh"
```

is the result of the XPath query.

A subtle point is that the value of the `starsOf` attribute in line (19) is itself a sequence of items — strings `"cf"` and `"mh"`. XPath expands sequences that are part of other sequences, so all items are at the “top level,” as we showed above. That is, a sequence of items is not itself an item. □

### 12.1.9 Conditions in Path Expressions

As we evaluate a path expression, we can restrict ourselves to follow only a subset of the paths whose tags match the tags in the expression. To do so, we follow a tag by a condition, surrounded by square brackets. This condition can be anything that has a boolean value. Values can be compared by comparison operators such as `=` or `>=`. “Not equal” is represented as in C, by `!=`. A compound condition can be constructed by connecting comparisons with operators `or` or `and`.

The values compared can be path expressions, in which case we are comparing the sequences returned by the expressions. Comparisons have an implied

“there exists” sense; two sequences are related if any pair of items, one from each sequence, are related by the given comparison operator. An example should make this concept clear.

**Example 12.6:** The following path expression:

```
/StarMovieData/Star[//City = "Malibu"]/Name
```

returns the names of the movie stars who have at least one home in Malibu. To begin, the path expression `/StarMovieData/Star` returns a sequence of all the **Star** elements. For each of these elements, we need to evaluate the truth of the condition `//City = "Malibu"`. Here, `//City` is a path expression, but it, like any path expression in a condition, is evaluated relative to the element to which the condition is applied. That is, we interpret the expression assuming that the element were the entire document to which the path expression is applied.

We start with the element for Carrie Fisher, lines (3) through (13) of Fig. 12.2. The expression `//City` causes us to look for all subelements, nested zero or more levels deep, that have a **City** tag. There are two, at lines (7) and (11). The result of the path expression `//City` applied to the Carrie-Fisher element is thus the sequence:

```
<City>Hollywood</City>
<City>Malibu</City>
```

Each item in this sequence is compared with the value `"Malibu"`. An element whose type is a primitive value such as a string can be equated to that string, so the second item passes the test. As a result, the entire **Star** element of lines (3) through (13) satisfies the condition.

When we apply the condition to the second item, lines (14) through (18) for Mark Hamill, we find a **City** subelement, but its value does not match `"Malibu"` and this element fails the condition. Thus, only the Carrie-Fisher element is in the result of the path expression

```
/StarMovieData/Star[//City = "Malibu"]
```

We have still to finish the XPath query by applying to this sequence of one element the continuation of the path expression, `/Name`. At this stage, we search for a **Name** subelement of the Carrie-Fisher element and find it at line (4). Consequently, the query result is the sequence of one element, `<Name>Carrie Fisher</Name>`. □

Several other useful forms of condition are:

- An integer  $[i]$  by itself is true only when applied the  $i$ th child of its parent.
- A tag  $[T]$  by itself is true only for elements that have one or more subelements with tag  $T$ .

- Similarly, an attribute  $[A]$  by itself is true only for elements that have a value for the attribute  $A$ .

**Example 12.7:** Figure 12.3 is a variant of our running movie example, in which we have grouped all the movies with a common title as one `Movie` element, with subelements that have tag `Version`. The title is an attribute of the movie, and the year is an attribute of the version. Versions have `Star` subelements. Consider the XPath query, applied to this document:

```
/Movies/Movie/Version[1]/@year
```

It asks for the year in which the first version of each movie was made, and the result is the sequence "1933" "1984".

```

1) <? xml version="1.0" encoding="utf-8" standalone="yes" ?>
2) <Movies>
3)   <Movie title = "King Kong">
4)     <Version year = "1933">
5)       <Star>Fay Wray</Star>
6)     </Version>
7)     <Version year = "1976">
8)       <Star>Jeff Bridges</Star>
9)       <Star>Jessica Lange</Star>
10)      </Version>
11)      <Version year = "2005" />
12)    </Movie>
13)    <Movie title = "Footloose">
14)      <Version year = "1984">
15)        <Star>Kevin Bacon</Star>
16)        <Star>John Lithgow</Star>
17)        <Star>Sarah Jessica Parker</Star>
18)      </Version>
19)    </Movie>
20)  </Movies>
```

Figure 12.3: An XML document for applying path expressions

In more detail, there are four `Version` elements that match the path

```
/Movies/Movie/Version
```

These are at lines (4) through (6), (7) through (10), line (11), and lines (14) through (18), respectively. Of these, the first and last are the first children of their respective parents. The `year` attributes for these versions are 1933 and 1984, respectively.  $\square$

**Example 12.8:** The XPath query:

```
/Movies/Movie/Version[Star]
```

applied to the document of Fig. 12.3 returns three **Version** elements. The condition **[Star]** is interpreted as “has at least one **Star** subelement.” That condition is true for the **Version** elements of lines (4) through (6), (7) through (10), and (14) through (18); it is false for the element of line (11). □

```
<Products>
  <Maker name = "A">
    <PC model = "1001" price = "2114">
      <Speed>2.66</Speed>
      <RAM>1024</RAM>
      <HardDisk>250</HardDisk>
    </PC>
    <PC model = "1002" price = "995">
      <Speed>2.10</Speed>
      <RAM>512</RAM>
      <HardDisk>250</HardDisk>
    </PC>
    <Laptop model = "2004" price = "1150">
      <Speed>2.00</Speed>
      <RAM>512</RAM>
      <HardDisk>60</HardDisk>
      <Screen>13.3</Screen>
    </Laptop>
    <Laptop model = "2005" price = "2500">
      <Speed>2.16</Speed>
      <RAM>1024</RAM>
      <HardDisk>120</HardDisk>
      <Screen>17.0</Screen>
    </Laptop>
  </Maker>
```

Figure 12.4: XML document with product data — beginning

### 12.1.10 Exercises for Section 12.1

**Exercise 12.1.1:** Figures 12.4 and 12.5 are the beginning and end, respectively, of an XML document that contains some of the data from our running products exercise. Write the following XPath queries. What is the result of each?

```
<Maker name = "E">
    <PC model = "1011" price = "959">
        <Speed>1.86</Speed>
        <RAM>2048</RAM>
        <HardDisk>160</HardDisk>
    </PC>
    <PC model = "1012" price = "649">
        <Speed>2.80</Speed>
        <RAM>1024</RAM>
        <HardDisk>160</HardDisk>
    </PC>
    <Laptop model = "2001" price = "3673">
        <Speed>2.00</Speed>
        <RAM>2048</RAM>
        <HardDisk>240</HardDisk>
        <Screen>20.1</Screen>
    </Laptop>
    <Printer model = "3002" price = "239">
        <Color>false</Color>
        <Type>laser</Type>
    </Printer>
<Maker name = "H">
    <Printer model = "3006" price = "100">
        <Color>true</Color>
        <Type>ink-jet</Type>
    </Printer>
    <Printer model = "3007" price = "200">
        <Color>true</Color>
        <Type>laser</Type>
    </Printer>
</Maker>
</Products>
```

Figure 12.5: XML document with product data — end

- a) Find the amount of RAM on each PC.
- b) Find the price of each product of any kind.
- c) Find all the printer elements.
- ! d) Find the makers of laser printers.
- ! e) Find the makers of PC's and/or laptops.
- f) Find the model numbers of PC's with a hard disk of at least 200 gigabytes.
- !! g) Find the makers of at least two PC's.

**Exercise 12.1.2:** The document of Fig. 12.6 contains data similar to that used in our running battleships exercise. In this document, data about ships is nested within their class element, and information about battles appears inside each ship element. Write the following queries in XPath. What is the result of each?

- a) Find the names of all ships.
- b) Find all the **Class** elements for classes with a displacement larger than 35000.
- c) Find all the **Ship** elements for ships that were launched before 1917.
- d) Find the names of the ships that were sunk.
- ! e) Find the years in which ships having the same name as their class were launched.
- ! f) Find the names of all ships that were in battles.
- !! g) Find the **Ship** elements for all ships that fought in two or more battles.

## 12.2 XQuery

XQuery is an extension of XPath that has become a standard for high-level querying of databases containing data in XML form. This section will introduce some of the important capabilities of XQuery.

```
<Ships>
  <Class name = "Kongo" type = "bc" country = "Japan"
    numGuns = "8" bore = "14" displacement = "32000">
    <Ship name = "Kongo" launched = "1913" />
    <Ship name = "Hiei" launched = "1914" />
    <Ship name = "Kirishima" launched = "1915">
      <Battle outcome = "sunk">Guadalcanal</Battle>
    </Ship>
    <Ship name = "Haruna" launched = "1915" />
  </Class>
  <Class name = "North Carolina" type = "bb" country = "USA"
    numGuns = "9" bore = "16" displacement = "37000">
    <Ship name = "North Carolina" launched = "1941" />
    <Ship name = "Washington" launched = "1941">
      <Battle outcome = "ok">Guadalcanal</Battle>
    </Ship>
  </Class>
  <Class name = "Tennessee" type = "bb" country = "USA"
    numGuns = "12" bore = "14" displacement = "32000">
    <Ship name = "Tennessee" launched = "1920">
      <Battle outcome = "ok">Surigao Strait</Battle>
    </Ship>
    <Ship name = "California" launched = "1921">
      <Battle outcome = "ok">Surigao Strait</Battle>
  </Class>
  <Class name = "King George V" type = "bb"
    country = "Great Britain"
    numGuns = "10" bore = "14" displacement = "32000">
    <Ship name = "King George V" launched = "1940" />
    <Ship name = "Prince of Wales" launched = "1941">
      <Battle outcome = "damaged">Denmark Strait</Battle>
      <Battle outcome = "sunk">Malaya</Battle>
    </Ship>
    <Ship name = "Duke of York" launched = "1941">
      <Battle outcome = "ok">North Cape</Battle>
    </Ship>
    <Ship name = "Howe" launched = "1942" />
    <Ship name = "Anson" launched = "1942" />
  </Class>
</Ships>
```

Figure 12.6: XML document containing battleship data

## Case Sensitivity of XQuery

XQuery is case sensitive. Thus, keywords such as `let` or `for` need to be written in lower case, just like keywords in C or Java.

### 12.2.1 XQuery Basics

XQuery uses the same model for values that we introduced for XPath in Section 12.1.1. That is, all values produced by XQuery expressions are sequences of items. Items are either primitive values or nodes of various types, including elements, attributes, and documents. Elements in a sequence are assumed to exist in the context of some document, as discussed in Section 12.1.7.

XQuery is a *functional language*, which implies that any XQuery expression can be used in any place that an expression is expected. This property is a very strong one. SQL, for example, allows subqueries in many places; but SQL does not permit, for example, any subquery to be any operand of any comparison in a where-clause. The functional property is a double-edged sword. It requires every operator of XQuery to make sense when applied to lists of more than one item, leading to some unexpected consequences.

To start, every XPath expression is an XQuery expression. There is, however, much more to XQuery, including FLWR (pronounced “flower”) expressions, which are in some sense analogous to SQL select-from-where expressions.

### 12.2.2 FLWR Expressions

Beyond XPath expressions, the most important form of XQuery expression involves clauses of four types, called for-, let-, where-, and return- (FLWR) clauses.<sup>1</sup> We shall introduce each type of clause in turn. However, we should be aware that there are options in the order and occurrences of these clauses.

1. The query begins with zero or more for- and let-clauses. There can be more than one of each kind, and they can be interlaced in any order, e.g., for, for, let, for, let.
2. Then comes an optional where-clause.
3. Finally, there is exactly one return-clause.

**Example 12.9:** Perhaps the simplest FLWR expression is:

```
return <Greeting>Hello World</Greeting>
```

It examines no data, and produces a value that is a simple XML element. □

---

<sup>1</sup>There is also an order-by clause that we shall introduce in Section 12.2.10. For that reason, FLWR is a less common acronym for the principal form of XQuery query than is FLWOR.

## Let Clauses

The simple form of a let-clause is:

*let variable := expression*

The intent of this clause is that the expression is evaluated and assigned to the variable for the remainder of the FLWR expression. Variables in XQuery must begin with a dollar-sign. Notice that the assignment symbol is `:=`, not an equal sign (which is used, as in XPath, in comparisons). More generally, a comma-separated list of assignments to variables can appear where we have shown one.

**Example 12.10:** One use of let-clauses is to assign a variable to refer to one of the documents whose data is used by the query. For example, if we want to query a document in file `stars.xml`, we can start our query with:

```
let $stars := doc("stars.xml")
```

In what follows, the value of `$stars` is a single doc node. It can be used in front of an XPath expression, and that expression will apply to the XML document contained in the file `stars.xml`. □

## For Clauses

The simple form of a for-clause is:

*for variable in expression*

The intent is that the expression is evaluated. The result of any expression is a sequence of items. The variable is assigned to each item, in turn, and what follows this for-clause in the query is executed once for each value of the variable. You will not be much deceived if you draw an analogy between an XQuery for-clause and a C for-statement. More generally, several variables may be set ranging over different sequences of items in one for-clause.

**Example 12.11:** We shall use the data suggested in Fig. 12.7 for a number of examples in this section. The data consists of two files, `stars.xml` in Fig. 12.7(a) and `movies.xml` in Fig. 12.7(b). Each of these files has data similar to what we used in Section 12.1, but the intent is that what is shown is just a small sample of the actual contents of these files.

Suppose we start a query:

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
    ... something done with each Movie element
```

```

1)  <? xml version="1.0" encoding="utf-8" standalone="yes" ?>
2)  <Stars>
3)      <Star>
4)          <Name>Carrie Fisher</Name>
5)          <Address>
6)              <Street>123 Maple St.</Street>
7)              <City>Hollywood</City>
8)          </Address>
9)          <Address>
10)             <Street>5 Locust Ln.</Street>
11)             <City>Malibu</City>
12)         </Address>
13)     </Star>
           ... more stars
14) </Stars>

```

(a) Document **stars.xml**

```

15) <? xml version="1.0" encoding="utf-8" standalone="yes" ?>
16) <Movies>
17)     <Movie title = "King Kong">
18)         <Version year = "1933">
19)             <Star>Fay Wray</Star>
20)         </Version>
21)         <Version year = "1976">
22)             <Star>Jeff Bridges</Star>
23)             <Star>Jessica Lange</Star>
24)         </Version>
25)         <Version year = "2005" />
26)     </Movie>
27)     <Movie title = "Footloose">
28)         <Version year = "1984">
29)             <Star>Kevin Bacon</Star>
30)             <Star>John Lithgow</Star>
31)             <Star>Sarah Jessica Parker</Star>
32)         </Version>
33)     </Movie>
           ... more movies
34) </Movies>

```

(b) Document **movies.xml**

## Boolean Values in XQuery

A comparision like `$x = 10` evaluates to true or false (strictly speaking, to one of the names `xs:true` or `xs:false` from the namespace for XML Schema). However, several other types of expressions can be interpreted as true or false, and so can serve as the value of a condition in a where-clause. The important coercions to remember are:

1. If the value is a sequence of items, then the empty sequence is interpreted as false and nonempty sequences as true.
2. Among numbers, 0 and `NAN` (“not a number,” in essence an infinite number) are false, and other numbers are true.
3. Among strings, the empty string is false and other strings are true.

Notice that `$movies/Movies/Movie` is an XPath expression that tells us to start with the document in file `movies.xml`, then go to the root `Movies` element, and then form the sequence of all `Movie` subelements. The body of the “for-loop” will be executed first with `$m` equal to the element of lines (17) through (26) of Fig. 12.7, then with `$m` equal to the element of lines (27) through (33), and then with each of the remaining `Movie` elements in the document. □

### The Where Clause

The form of a where-clause is:

*where condition*

This clause is applied to an item, and the *condition*, which is an expression, evaluates to true or false. If the value is true, then the return-clause is applied to the current values of any variables in the query. Otherwise, nothing is produced for the current values of variables.

### The Return Clause

The form of this clause is:

*return expression*

The result of a FLWR expression, like that of any expression in XQuery, is a sequence of items. The sequence of items produced by the expression in the return-clause is appended to the sequence of items produced so far. Note that although there is only one return-clause, this clause may be executed many

times inside “for-loops,” so the result of the query may be constructed in stages. We should not think of the return-clause as a “return-statement,” since it does not end processing of the query.

**Example 12.12:** Let us complete the query we started in Example 12.11 by asking for a list of all the star elements found among the versions of all movies. The query is:

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
    return $m/Version/Star
```

The first value of `$m` in the “for-loop” is the element of lines (17) through (26) of Fig. 12.7. From that `Movie` element, the XPath expression `/Version/Star` produces a sequence of the three `Star` elements at lines (19), (22), and (23). That sequence begins the result of the query.

```
<Star>Fay Wray</Star>
<Star>Jeff Bridges</Star>
<Star>Jessica Lange</Star>
<Star>Kevin Bacon</Star>
<Star>John Lithgow</Star>
<Star>Sarah Jessica Parker</Star>
...
...
```

Figure 12.8: Beginning of the result sequence for the query of Example 12.12

The next value of `$m` is the element of lines (27) through (33). Now, the result of the expression in the return-clause is the sequence of elements in lines (29), (30), and (31). Thus the beginning of the result sequence looks like that in Fig. 12.8. □

### 12.2.3 Replacement of Variables by Their Values

Let us consider a modification to the query of Example 12.12. Here, we want to produce not just a sequence of `<Star>` elements, but rather a sequence of `Movie` elements, each containing all the stars of movies with a given title, regardless of which version they starred in. The title will be an attribute of the `Movie` element.

Figure 12.9 shows an attempt that seems right, but in fact *is not correct*. The expression we return for each value of `$m` seems to be an opening `<Movie>` tag followed by the sequence of `Star` elements for that movie, and finally a closing `</Movie>` tag. The `<Movie>` tag has a `title` attribute that is a copy of the same attribute from the `Movie` element in file `movies.xml`. However, when we execute this program, what appears is:

## Sequences of Sequences

We should remind the reader that sequences of items can have no internal structure. Thus, in Fig. 12.8, there is no separator between Jessica Lange and Kevin Bacon, or any grouping of the first three stars and the last three, even though these groups were produced by different executions of the return-clause.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title = "$m/@title">$m/Version/Star</Movie>
```

Figure 12.9: Erroneous attempt to produce `Movie` elements

```
<Movie title = "$m/@title">$m/Version/Star</Movie>
<Movie title = "$m/@title">$m/Version/Star</Movie>
...

```

The problem is that, between tags, or as the value of an attribute, any text string is permissible. This return statement looks no different, to the XQuery processor, than the return of Example 12.9, where we really were producing text inside matching tags. In order to get text interpreted as XQuery expressions inside tags, we need to surround the text by curly braces.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title = "{$m/@title}>{$m/Version/Star}</Movie>
```

Figure 12.10: Adding curly braces fixes the problem

The proper way to meet our goal is shown in Fig. 12.10. In this query, the expressions `$m/title` and `$m/Version/Star` inside the braces are properly interpreted as XPath expressions. The first is replaced by a text string, and the second is replaced by a sequence of `Star` elements, as intended.

**Example 12.13 :** This example not only further illustrates the use of curly braces to force interpretation of expressions, but also emphasizes how any XQuery expression can be used wherever an expression of any kind is permitted. Our goal is to duplicate the result of Example 12.12, where we got a sequence of `Star` elements, but to make the entire sequence of stars be within a `Stars` element. We cannot use the trick of Fig. 12.10 with `Stars` in place of `Star`, because that would place many `Stars` tags around separate groups of stars.

```

let $starSeq := (
    let $movies := doc("movies.xml")
    for $m in $movies/Movies/Movie
        return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>

```

Figure 12.11: Putting tags around a sequence

Figure 12.11 does the job. We assign the sequence of **Star** elements that results from the query of Example 12.12 to a local variable **\$starSeq**. We then return that sequence, surrounded by tags, being careful to enclose the variable in braces so it is evaluated and not treated literally. □

#### 12.2.4 Joins in XQuery

We can join two or more documents in XQuery in much the same way as we join two or more relations in SQL. In each case we need variables, each of which ranges over elements of one of the documents or tuples of one of the relations, respectively. In SQL, we use a *from-clause* to introduce the needed tuple variables (which may just be the table name itself); in XQuery we use a *for-clause*.

However, we must be very careful how we do comparisons in a join. First, there is the matter of comparison operators such as = or < operating on sequences with the meaning of “there exist elements that compare” as discussed in Section 12.1.9. We shall take up this point again in Section 12.2.5. Additionally, equality of elements is by “element identity” (analogous to “object identity”). That is, an element is not equal to a different element, even if it looks the same, character-by-character. Fortunately, we usually do not want to compare elements, but really the primitive values such as strings and integers that appear as values of their attributes and subelements. The comparison operators work as expected on primitive values; < is “precedes in lexicographic order” for strings.

There is a built-in function **data(E)** that extracts the value of an element *E*. We can use this function to extract the text from an element that is a string with matching tags.

**Example 12.14:** Suppose we want to find the cities in which stars mentioned in the **movies.xml** file of Fig. 12.7(b) live. We need to consult the **stars.xml** file of Fig. 12.7(a) to get that information. Thus, we set up a variable ranging over the **Star** elements of **movies.xml** and another variable ranging over the **Star** elements of **stars.xml**. When the data in a **Star** element of **movies.xml** matches the data in the **Name** subelement of a **Star** element of **stars.xml**, then we have a match, and we extract the **City** element of the latter.

Figure 12.12 shows a solution. The let-clause introduces variables to stand for the two documents. As before, this shorthand is not necessary, and we could have used the document nodes themselves in the XPath expressions of the next two lines. The for-clause introduces a doubly nested loop. Variable `$s1` ranges over each `Star` element of `movies.xml` and `$s2` does the same for `stars.xml`.

```
let $movies := doc("movies.xml"),
    $stars := doc("stars.xml")
for $s1 in $movies/Movies/Movie/Version/Star,
    $s2 in $stars/Stars/Star
where data($s1) = data($s2/Name)
return $s2/Address/City
```

Figure 12.12: Finding the cities of stars

The where-clause uses the built-in function `data` to extract the strings that are the values of the elements `$s1` and `$s2`. Finally, the return-clause produces a `City` element. □

### 12.2.5 XQuery Comparison Operators

We shall now consider another puzzle where things don't quite work as expected. Our goal is to find the stars in `stars.xml` of Fig. 12.7(a) that live at 123 Maple St., Malibu. Our first attempt is in Fig. 12.13.

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $s/Address/Street = "123 Maple St." and
      $s/Address/City = "Malibu"
return $s/Name
```

Figure 12.13: An erroneous attempt to find who lives at 123 Maple St., Malibu

In the where-clause, we compare `Street` elements and `City` elements with strings, but that works as expected, because an element whose value is a string is coerced to that string, and the comparison will succeed when expected. The problem is seen when `$s` takes the `Star` element of lines (3) through (13) of Fig. 12.7 as its value. Then, XPath expression `$s/Address/Street` produces the sequence of two elements of lines (6) and (10) as its value. Since the `=` operator returns true if any pair of items, one from each side, equate, the value of the first condition is true; line (6), after coercion, is equal to the string "123 Maple St.". Similarly, the second condition compares the list of two `City` elements of lines (7) and (11) with the string "Malibu", and equality is found for line (11). As a result, the `Name` element for Carrie Fisher [line (4)] is returned.

But Carrie Fisher doesn't live at 123 Maple St., Malibu. She lives at 123 Maple St., Hollywood, and elsewhere in Malibu. The existential nature of comparisons has caused us to fail to notice that we were getting a street and city from different addresses.

XQuery provides a set of comparison operators that only compare sequences consisting of a single item, and fail if either operand is a sequence of more than one item. These operators are two-letter abbreviations for the comparisons: `eq`, `ne`, `lt`, `gt`, `le`, and `ge`. We could use `eq` in place of `=` to catch the case where we are actually comparing a string with several streets or cities. The revised query is shown in Fig. 12.14.

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $s/Address/Street eq "123 Maple St." and
      $s/Address/City eq "Malibu"
return $s/Name
```

Figure 12.14: A second erroneous attempt to find who lives at 123 Maple St., Malibu

This query does not allow the Carrie-Fisher element to pass the test of the where-clause, because the left sides of the `eq` operator are not single items, and therefore the comparison fails. Unfortunately, it will not report any star with two or more addresses, even if one of those addresses is 123 Maple St., Malibu. Writing a correct query is tricky, regardless of which version of the comparison operators we use, and we leave a correct query as an exercise.

### 12.2.6 Elimination of Duplicates

XQuery allows us to eliminate duplicates in sequences of any kind, by applying the built-in function `distinct-values`. There is a subtlety that must be noted, however. Strictly speaking, `distinct-values` applies to primitive types. It will strip the tags from an element that is a tagged text-string, but it won't put them back. Thus, the input to `distinct-values` can be a list of elements and the result a list of strings.

**Example 12.15:** Figure 12.11 gathered all the `Star` elements from all the movies and returned them as a sequence. However, a star that appeared in several movies would appear several times in the sequence. By applying `distinct-values` to the result of the subquery that becomes the value of variable `$starseq`, we can eliminate all but one copy of each `Star` element. The new query is shown in Fig. 12.15.

Notice, however, that what is produced is a list of the names of the stars surrounded by the `Stars` tags, as:

```
<Stars>"Fay Wray" "Jeff Bridges" ... </Stars>
```

```

let $starSeq := distinct-values(
    let $movies := doc("movies.xml")
    for $m in $movies/Movies/Movie
    return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>

```

Figure 12.15: Eliminating duplicate stars

In comparison, the version in Fig. 12.11 produced

```

<Stars><Star>Fay Wray</Star> <Star>Jeff Bridges</Star> ...
</Stars>

```

but might produce duplicates. □

### 12.2.7 Quantification in XQuery

There are expressions that say, in effect, “for all” and “there exists.” Their forms, respectively, are:

*every variable in expression1 satisfies expression2*  
*some variable in expression1 satisfies expression2*

Here, *expression1* produces a sequence of items, and the variable takes on each item, in turn, as its value. For each such value, *expression2* (which normally involves the variable) is evaluated, and should produce a boolean value.

In the “every” version, the result of the entire expression is false if some item produced by *expression1* makes *expression2* false; the result is true otherwise. In the “some” version, the result of the entire expression is true if some item produced by *expression1* makes *expression2* true; the result is false otherwise.

```

let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where every $c in $s/Address/City satisfies
    $c = "Hollywood"
return $s/Name

```

Figure 12.16: Finding the stars who only live in Hollywood

**Example 12.16:** Using the data in the file **stars.xml** of Fig. 12.7(a), we want to find those stars who live in Hollywood and nowhere else. That is, no matter how many addresses they have, they all have city Hollywood. Figure 12.16 shows how to write this query. Notice that **\$s/Address/City** produces the

sequence of `City` elements of the star `$s`s. The where-clause is thus satisfied if and only if every element on that list is `<City>Hollywood</City>`.

Incidentally, we could change the “every” to “some” and find the stars that have at least one home in Hollywood. However, it is rarely necessary to use the “some” version, since most tests in XQuery are existentially quantified anyway. For instance,

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $s/Address/City = "Hollywood"
return $s/Name
```

produces the stars with a home in Hollywood, without using a “some” expression. Recall our discussion in Section 12.2.5 of how a comparision such as `=`, with a sequence of more than one item on either or both sides, is true if we can match any items from the two sides. □

### 12.2.8 Aggregations

XQuery provides built-in functions to compute the usual aggregations such as `count`, `sum`, or `max`. They take any sequence as argument; that is, they can be applied to the result of any XQuery expression.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
where count($m/Version) > 1
return $m
```

Figure 12.17: Finding the movies with multiple versions

**Example 12.17:** Let us examine the data in file `movies.xml` of Fig. 12.7(b) and produce those `Movie` elements that have more than one version. Figure 12.17 does the job. The XPath expression `$m/Version` produces the sequence of `Version` elements for the movie `$m`. The number of items in the sequence is counted. If that count exceeds 1, the where-clause is satisfied, and the movie element `$m` is appended to the result. □

### 12.2.9 Branching in XQuery Expressions

There is an if-then-else expression in XQuery of the form

$$\text{if } (\textit{expression}_1) \text{ then } \textit{expression}_2 \text{ else } \textit{expression}_3$$

To evaluate this expression, first evaluate `expression1`; if it is true, evaluate `expression2`, which becomes the result of the whole expression. If `expression1` is false, the result of the whole expression is `expression3`.

This expression is not a statement — there are no statements in XQuery, only expressions. Thus, the analog in C is the ?: expression, not the if-then-else statement. Like the expression in C, there is no way to omit the “else” part. However, we can use as *expression3* the empty sequence, which is denoted (). This choice makes the conditional expression produce the empty sequence when the test-condition is not satisfied.

**Example 12.18:** Our goal in this example is to produce each of the versions of *King Kong*, tagging the most recent version **Latest** and the earlier versions **Old**. In line (1), we set variable \$kk to be the **Movie** element for *King Kong*. Notice that we have used an XPath condition in this line, to make sure that we produce only that one element. Of course, if there were several **Movie** elements that had the title *King Kong*, then all of them would be on the sequence of items that is the value of \$kk, and the query would make no sense. However, we are assuming title is a key for movies in this structure, since we have explicitly grouped versions of movies with the same title.

```

1) let $kk := doc("movies.xml")/Movies/Movie[@title = "King Kong"]
2) for $v in $kk/Version
3) return
4)   if ($v/@year = max($kk/Version/@year))
5)     then <Latest>{$v}</Latest>
6)     else <Old>{$v}</Old>
```

Figure 12.18: Tagging the versions of *King Kong*

Line (2) causes \$v to iterate over all versions of *King Kong*. For each such version, we return one of two elements. To tell which, we evaluate the condition of line (4). On the right of the equal-sign is the maximum year of any of the *King-Kong* versions, and on the left is the year of the version \$v. If they are equal, then \$v is the latest version, and we produce the element of line (5). If not, then \$v is an old version, and we produce the element of line (6). □

### 12.2.10 Ordering the Result of a Query

It is possible to sort the results as part of a FLWR query, if we add an order-clause before the return-clause. In fact, the query form we have been concentrating on here is usually called FLWOR (but still pronounced “flower”), to acknowledge the optional presence of an order-clause. The form of this clause is:

*order list of expressions*

The sort is based on the value of the first expression, ties are broken by the value of the second expression, and so on. The default order is ascending, but the keyword **descending** following an expression reverses the order.

What happens when an order is present is analogous to what happens in SQL. Just before we reach the stage in query processing where the output is assembled (the **SELECT** clause in SQL; the return-clause in XQuery), the result of previous clauses is assembled and sorted. In the case of SQL, the intermediate result is a set of bindings of tuples to the tuple variables that range over each of the relations in the **FROM** clause. Specifically, it is all those bindings that pass the test of the **WHERE** clause.

In XQuery, we should think of the intermediate result as a sequence of bindings of variables to values. The variables are those defined in the for- and let-clauses that precede the order-clause, and the sequence consists of all those bindings that pass the test of the where-clause. These bindings are each used to evaluate the expressions in the order-clause, and the values of those expressions govern the position of the binding in the order of all the bindings. Once we have the order of bindings, we use them, in turn, to evaluate the expression in the return-clause.

**Example 12.19:** Let us consider all versions of all movies, order them by year, and produce a sequence of **Movie** elements with the title and year as attributes. The data comes from file **movies.xml** in Fig. 12.7(b), as usual. The query is shown in Fig. 12.19.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie,
    $v in $m/Version
order $v/@year
return <Movie title = "{$m/@title}" year = "{$v/@year}" />
```

Figure 12.19: Construct the sequence of title-year pairs, ordered by year

When we reach the order-clause, bindings provide values for the three variables **\$movies**, **\$m**, and **\$v**. The value **doc("movies.xml")** is bound to **\$movies** in every one of these bindings. However, the values of **\$m** and **\$v** vary; for each pair consisting of a movie and a version of that movie, there will be one binding for the two variables. For instance, the first such binding associates with **\$m** the element in lines (17) through (26) of Fig. 12.7(b) and associates with **\$v** the element of lines (18) through (20).

The bindings are sorted according to the value of attribute **year** in the element to which **\$v** is bound. There may be many movies with the same year, and the ordering does not specify how these are to be ordered. As a result, all we know is that the movie-version pairs with a given year will appear together in some order, and the groups for each year will be in the ascending order of year. If we wanted to specify a total ordering of the bindings, we could, for example, add a second term to the list in the order-clause, such as:

```
order $v/@year, $m/@title
```

to break ties alphabetically by title.

After sorting the bindings, each binding is passed to the return-clause, in the order chosen. By substituting for the variables in the return-clause, we produce from each binding a single `Movie` element. □

### 12.2.11 Exercises for Section 12.2

**Exercise 12.2.1:** Using the product data from Figs. 12.4 and 12.5, write the following in XQuery.

- a) Find the `Printer` elements with a price less than 100.
- b) Find the `Printer` elements with a price less than 100, and produce the sequence of these elements surrounded by a tag `<CheapPrinters>`.
- ! c) Find the names of the makers of both printers and laptops.
- ! d) Find the names of the makers that produce at least two `PC`'s with a speed of 3.00 or more.
- ! e) Find the makers such that every `PC` they produce has a price no more than 1000.
- !! f) Produce a sequence of elements of the form

```
<Laptop><Model>x</Model><Maker>y</Maker></Laptop>
```

where  $x$  is the model number and  $y$  is the name of the maker of the laptop.

**Exercise 12.2.2:** Using the battleships data of Fig. 12.6, write the following in XQuery.

- a) Find the names of the classes that had at least 10 guns.
- b) Find the names of the ships that had at least 10 guns.
- c) Find the names of the ships that were sunk.
- d) Find the names of the classes with at least 3 ships.
- ! e) Find the names of the classes such that no ship of that class was in a battle.
- !! f) Find the names of the classes that had at least two ships launched in the same year.
- !! g) Produce a sequence of items of the form

```
<Battle name = x><Ship name = y />...</Battle>
```

where  $x$  is the name of a battle and  $y$  the name of a ship in the battle. There may be more than one `Ship` element in the sequence.

- ! **Exercise 12.2.3:** Solve the problem of Section 12.2.5; write a query that finds the star(s) living at a given address, even if they have several addresses, without finding stars that do not live at that address.
- ! **Exercise 12.2.4:** Do there exist expressions  $E$  and  $F$  such that the expression `every $x in E satisfies F` is true, but `some $x in E satisfies F` is false? Either give an example or explain why it is impossible.

## 12.3 Extensible Stylesheet Language

XSLT (Extensible Stylesheet Language for Transformations) is a standard of the World-Wide-Web Consortium. Its original purpose was to allow XML documents to be transformed into HTML or similar forms that allowed the document to be viewed or printed. However, in practice, XSLT is another query language for XML. Like XPath or XQuery, we can use XSLT to extract data from documents or turn one document form into another form.

### 12.3.1 XSLT Basics

Like XML Schema, XSLT specifications are XML documents; these specifications are usually called *stylesheets*. The tags used in XSLT are found in a namespace, which is `http://www.w3.org/1999/XSL/Transform`. Thus, at the highest level, a stylesheet looks like Fig. 12.20.

```
<? xml version = "1.0" encoding = "utf-8" ?>
<xsl:stylesheet xmlns:xsl =
    "http://www.w3.org/1999/XSL/Transform">
    ...
</xsl:stylesheet>
```

Figure 12.20: The form of an XSLT stylesheet

### 12.3.2 Templates

A stylesheet will have one or more *templates*. To apply a stylesheet to an XML document, we go down the list of templates until we find one that matches the root. As processing proceeds, we often need to find matching templates for elements nested within the document. If so, we again search the list of templates for a match according to matching rules that we shall learn in this section. The simplest form of a template tag is:

```
<xsl:template match = "XPath expression">
```

The XPath expression, which can be either rooted (beginning with a slash) or relative, describes the elements of an XML document to which this template is applied. If the expression is rooted, then the template is applied to every element of the document that matches the path. Relative expressions are applied when a template  $T$  has within it a tag `<xsl:apply-templates>`. In that case, we look among the children of the elements to which  $T$  is applied. In that way, we can traverse an XML document's tree in a depth-first manner, performing complicated transformations on the document.

The simplest content of a template is text, typically HTML. When a template matches a document, the text inside that document is produced as output. Within the text can be calls to apply templates to the children and/or obtain values from the document itself, e.g., from attributes of the current element.

```

1)   <? xml version = "1.0" encoding = "utf-8" ?>
2)   <xsl:stylesheet xmlns:xsl =
3)           "http://www.w3.org/1999/XSL/Transform">
4)       <xsl:template match = "/">
5)           <HTML>
6)               <BODY>
7)                   <B>This is a document</b>
8)               </body>
9)           </html>
10)      </xsl:template>
11)  </xsl:stylesheet>
```

Figure 12.21: Printing output for any document

**Example 12.20 :** In Fig. 12.21 is an exceedingly simple stylesheet. It applies to any document and produces the same HTML document, regardless of its input. This HTML document says “This is a document” in boldface.

Line (4) introduces the one template in the stylesheet. The value of the `match` attribute is `"/"`, which matches only the root. The body of the template, lines (5) through (9), is simple HTML. When these lines are produced as output, the resulting file can be treated as HTML and displayed by a browser or other HTML processor.  $\square$

### 12.3.3 Obtaining Values From XML Data

It is unusual that the document we produce does not depend in any way on the input to the transformation, as was the case in Example 12.20. The simplest way to extract data from the input is with the `value-of` tag. The form of this tag is:

```

<? xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Movies>
    <Movie title = "King Kong">
        <Version year = "1933">
            <Star>Fay Wray</Star>
        </Version>
        <Version year = "1976">
            <Star>Jeff Bridges</Star>
            <Star>Jessica Lange</Star>
        </Version>
        <Version year = "2005" />
    </Movie>
    <Movie title = "Footloose">
        <Version year = "1984">
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parker</Star>
        </Version>
    </Movie>
    ...
</Movies>

```

Figure 12.22: The file `movies.xml`

```
<xsl:value-of select = "expression" />
```

The expression is an XPath expression that should produce a string as value. Other values, such as elements containing text, are coerced into strings in the obvious way.

**Example 12.21:** In Fig. 12.22 we reproduce the file `movies.xml` that was used in Section 12.2 as a running example. In this example of a stylesheet, we shall use `value-of` to obtain all the titles of movies and print them, one to a line. The stylesheet is shown in Fig. 12.23.

At line (4), we see that the template matches every `Movie` element, so we process them one at a time. Line (5) applies the `value-of` operation with an XPath expression `@title`. That is, we go to the `title` attribute of each `Movie` element and take the value of that attribute. This value is produced as output, and followed at line (6) by the HTML break tag, so the next movie title will be printed on the next line. □

### 12.3.4 Recursive Use of Templates

The most interesting and powerful transformations require recursive application of templates at various elements of the input. Having selected a template to

```

1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xsl:stylesheet xmlns:xsl =
3)      "http://www.w3.org/1999/XSL/Transform">
4)      <xsl:template match = "/Movies/Movie">
5)          <xsl:value-of select = "@title" />
6)          <BR/>
7)      </xsl:template>
8)  </xsl:stylesheet>

```

Figure 12.23: Printing the titles of movies

apply to the root of the input document, we can ask that a template be applied to each of its subelements, by using the **apply-templates** tag. If we want to apply a certain template to only some subset of the subelements, e.g., those with a certain tag, we can use a **select** expression, as:

```
<xsl:apply-templates select = "expression" />
```

When we encounter such a tag within a template, we find the set of matching subelements of the current element (the element to which the template is being applied). For each subelement, we find the first template that matches and apply it to the subelement.

**Example 12.22 :** In this example, we shall use XSLT to transform an XML document into another XML document, rather than into an HTML document. Let us examine Fig. 12.24. There are four templates, and together they process movie data in the form of Fig. 12.22. The first template, lines (4) through (8), matches the root. It says to output the text **<Movies>** and then apply templates to the children of the root element. We could have specified that templates were to be applied only to children that are tagged **<Movie>**, but since we expect no other tags among the children, we did not specify:

```
6)      <xsl:apply-templates select = "Movie" />
```

Notice that after applying templates to the **<Movie>** children (which will result in the printing of many elements), we close the **<Movies>** element in the output with the appropriate closing tag at line (7). Also observe that we can tell the difference between tags that are output text, such as lines (5) and (7), from tags that are XSLT, because all XSLT tags must be from the **xsl** namespace.

Now, let us see what applying templates to the **<Movie>** elements does. The first (and only) template that matches these elements is the second, at lines (9) through (15). This template begins by outputting the text **<Movie title = "** at line (10). Then, line (11) obtains the title of the movie and emits it to the output. Line (12) finishes the quoted attribute value and the **<Movie>** tag in the output. Line (13) applies templates to all the children of the movie, which should be versions. Finally, line (14) emits the matching **</Movie>** ending tag.

```

1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xsl:stylesheet xmlns:xsl =
3)      "http://www.w3.org/1999/XSL/Transform">

4)      <xsl:template match = "/Movies">
5)          <Movies>
6)              <xsl:apply-templates />
7)          </Movies>
8)      </xsl:template>

9)      <xsl:template match = "Movie">
10)         <Movie title =
11)             <xsl:value-of select = "@title" />
12)         ">
13)             <xsl:apply-templates />
14)         </Movie>
15)     </xsl:template>

16)     <xsl:template match = "Version">
17)         <xsl:apply-templates />
18)     </xsl:template>

19)     <xsl:template match = "Star">
20)         <Star name =
21)             <xsl:value-of select = "." />
22)         " />
23)     </xsl:template>

24)     </xsl:stylesheet>

```

Figure 12.24: Transforming the `movies.xml` file

When line (13) calls for templates to be applied to all the versions of a movie, the only matching template is that of lines (16) through (18), which does nothing but apply templates to the children of the version, which should be `<Star>` elements. Thus, what gets generated between each opening `<Movie>` tag and its matched closing tag is determined by the last template of lines (19) through (23). This template is applied to each `<Star>` element.

Star elements from the input are transformed in the output. Instead of the star's name being text, as it is in Fig. 12.22, the template starting at line (19) produces a `<Star>` element with the name as an attribute. Line (21) says to select the `<Star>` element itself (the dot represents the “self” axis as an XPath expression) as a value for the output. However, all output is text, so the tags of the element are not part of the output. That result is exactly what we want,

since the value of the attribute `name` should be a string, not an element. The empty `<Star>` element is completed on line (22). For instance, given the input of Fig. 12.22, the output would be as shown in Fig. 12.25. □

```

<Movies>
  <Movie title = "King Kong">
    <Star name = "Fay Wray" />
    <Star name = "Jeff Bridges" />
    <Star name = "Jessica Lange" />
  </Movie>
  <Movie title = "Footloose">
    <Star name = "Kevin Bacon" />
    <Star name = "John Lithgow" />
    <Star name = "Sarah Jessica Parker" />
  </Movie>
  ...
</Movies>

```

Figure 12.25: Output of the transform of Fig. 12.24

### 12.3.5 Iteration in XSLT

We can put a loop within a template that gives us freedom over the order in which we visit certain subelements of the element to which the template is being applied. The `for-each` tag creates the loop, with a form:

```
<xsl:for-each select = "expression">
```

The expression is an XPath expression whose value is a sequence of items. Whatever is between the opening `<for-each>` tag and its matched closing tag is executed for each item, in turn.

**Example 12.23:** In Fig. 12.26 is a copy of our document `stars.xml`; we wish to transform it to an HTML list of all the names of stars followed by an HTML list of all the cities in which stars live. Figure 12.27 has a template that does the job.

There is one template, which matches the root. The first thing that happens is at line (5), where the HTML tag `<OL>` is emitted to start an ordered list. Then, line (6) starts a loop, which iterates over each `<Star>` subelement. At lines (7) through (9), a list item with the name of that star is emitted. Line (11) ends the list of names and begins a list of cities. The second loop, lines (12) through (16), runs through each `<Address>` element and emits a list item for the city. Line (17) closes the second list. □

```

<? xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Stars>
  <Star>
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maple St.</Street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ln.</Street>
      <City>Malibu</City>
    </Address>
  </Star>
  ... more stars
</Stars>

```

Figure 12.26: Document stars.xml

```

1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xsl:stylesheet xmlns:xsl =
3)      "http://www.w3.org/1999/XSL/Transform">
4)      <xsl:template match = "/">
5)          <OL>
6)              <xsl:for-each select = "Stars/Star" />
7)                  <LI>
8)                      <xsl:value-of select = "Name">
9)                  </li>
10)                 </xsl:for-each>
11)             </ol><P/><OL>
12)                 <xsl:for-each select = "Stars/Star/Address" />
13)                     <LI>
14)                         <xsl:value-of select = "City">
15)                     </li>
16)                 </xsl:for-each>
17)             </ol>
18)         </xsl:template>
19)     </xsl:stylesheet>

```

Figure 12.27: Printing names and cities of stars

### 12.3.6 Conditionals in XSLT

We can introduce branching into our templates by using an **if** tag. The form of this tag is:

```
<xsl:if test = "boolean expression">
```

Whatever appears between this tag and its matched closing tag is executed if and only if the boolean expression is true. There is no else-clause, but we can follow this expression by another **if** that has the opposite test condition should we wish.

```

1)   <? xml version = "1.0" encoding = "utf-8" ?>
2)   <xsl:stylesheet xmlns:xsl =
3)       "http://www.w3.org/1999/XSL/Transform">
4)   <xsl:template match = "/">
5)       <TABLE border = "5"><TR><TH>Stars</th></tr>
6)       <xsl:for-each select = "Stars/Star" />
7)           <xsl:if test = "Address/City = 'Hollywood'">
8)               <TR><TD>
9)                   <xsl:value-of select = "Name" />
10)                  </td></tr>
11)          </xsl:if>
12)      </xsl:for-each>
13)  </table>
14)  </xsl:template>
15) </xsl:stylesheet>
```

Figure 12.28: Finding the names of the stars who live in Hollywood

**Example 12.24:** Figure 12.28 is a stylesheet that prints a one-column table, with header “Stars.” There is one template, which matches the root. The first thing this template does is print the header row at line (5). The for-each loop of lines (6) through (12) iterates over each star. The conditional of line (7) tests whether the star has at least one home in Hollywood. Remember that the equal-sign represents a comparison is true if any item on the left equals any item on the right. That is what we want, since we asked whether any of the homes a star has is in Hollywood. Lines (8) through (10) print a row of the table. □

### 12.3.7 Exercises for Section 12.3

**Exercise 12.3.1:** Suppose our input XML document has the form of the product data of Figs. 12.4 and 12.5. Write XSLT stylesheets to produce each of the following documents.

- a) An HTML file consisting of a header “Manufacturers” followed by an enumerated list of the names of all the makers of products listed in the input.
- b) An HTML file consisting of a table with headers “Model” and “Price,” with a row for each PC. That row should have the proper model and price for the PC.
- ! c) An HTML file consisting of a table whose headers are “Model,” “Price,” “Speed,” and “Ram” for all Laptops, followed by another table with the same headers for PC’s.
- d) An XML file with root tag `<PCs>` and subelements having tag `<PC>`. This tag has attributes `model`, `price`, `speed`, and `ram`. In the output, there should be one `<PC>` element for each `<PC>` element of the input file, and the values of the attributes should be taken from the corresponding input element.
- !! e) An XML file with root tag `<Products>` whose subelements are `<Product>` elements. Each `<Product>` element has attributes `type`, `maker`, `model`, and `price`, where the type is one of "PC", "Laptop", or "Printer". There should be one `<Product>` element in the output for every PC, laptop, and printer in the input file, and the output values should be chosen appropriately from the input data.
- ! f) Repeat part (b), but make the output file a Latex file.

**Exercise 12.3.2:** Suppose our input XML document has the form of the product data of Fig. 12.6. Write XSLT stylesheets to produce each of the following documents.

- a) An HTML file with a header for each class. Under each header is a table with column-headers “Name” and “Launched” with the appropriate entry for each ship of the class.
- b) An HTML file with root tag `<Losers>` and subelements `<Ship>`, each of whose values is the name of one of the ships that were sunk.
- ! c) An XML file with root tag `<Ships>` and subelements `<Ship>` for each ship. These elements each should have attributes `name`, `class`, `country` and `numGuns` with the appropriate values taken from the input file.
- ! d) Repeat (c), but only list those ships that were in at least one battle.
- e) An XML file identical to the input, except that `<Battle>` elements should be empty, with the outcome and name of the battle as two attributes.

## 12.4 Summary of Chapter 12

- ◆ **XPath:** This language is a simple way to express many queries about XML data. You describe paths from the root of the document by sequences of tags. The path may end at an attribute rather than an element.
- ◆ **The XPath Data Model:** All XPath values are sequences of items. An item is either a primitive value or an element. An element is an opening XML tag, its matched closing tag, and everything in between.
- ◆ **Axes:** Instead of proceeding down the tree in a path, one can follow another axis, including jumps to any descendant, a parent, or a sibling.
- ◆ **XPath Conditions:** Any step in a path can be constrained by a condition, which is a boolean-valued expression. This expression appears in square brackets.
- ◆ **XQuery:** This language is a more advanced form of query language for XML documents. It uses the same data model as XPath. XQuery is a functional language.
- ◆ **FLWR Expressions:** Many queries in XQuery consist of let-, for-, where- and return-clauses. “Let” introduces temporary definitions of variables; “for” creates loops; “where” supplies conditions to be tested, and “return” defines the result of the query.
- ◆ **Comparison Operators in XQuery and XPath:** The conventional comparison operators such as `<` apply to sequences of items, and have a “there-exists” meaning. They are true if the stated relation holds between any pair of items, one from each of the lists. To be assured that single items are being compared, we can use letter codes for the operators, such as `lt` for “less than.”
- ◆ **Other XQuery Expressions:** XQuery has many operations that resemble those in SQL. These operators include existential and universal quantification, aggregation, duplicate-elimination, and sorting of results.
- ◆ **XSLT:** This language is designed for transformations of XML documents, although it also can be used as a query language. A “program” in this language has the form of an XML document, with a special namespace that allows us to use tags to describe a transformation.
- ◆ **Templates:** The heart of XSLT is a template, which matches certain elements of the input document. The template describes output text, and can extract values from the input document for inclusion in the output. A template can also call for templates to be applied recursively to the children of an element.

- ◆ *XSLT Programming Constructs:* A template can also include XSLT constructs that behave like an iterative programming language. These constructs include for-loops and if-statements.

## 12.5 References for Chapter 12

The World-Wide-Web Consortium site for the definition of XPath is [2]. The site for XQuery is [3], and for XSLT it is [4].

[1] is an introduction to the XQuery language. There are tutorials for XPath, XQuery, and XSLT at [5].

1. D. D. Chamberlin, “XQuery: an XML Query Language,” *IBM Systems Journal* 41:4 (2002), pp. 597–615. See also  
[www.research.ibm.com/journal/sj/414/chamberlin.pdf](http://www.research.ibm.com/journal/sj/414/chamberlin.pdf)
2. World-Wide-Web Consortium <http://www.w3.org/TR/xpath>
3. World-Wide-Web Consortium <http://www.w3.org/TR/xquery>
4. World-Wide-Web Consortium <http://www.w3.org/TR/xslt>
5. W3 Schools, <http://www.w3schools.com>

# Index

## A

Abiteboul, S. 12, 515

Abort

    See Rollback

Achilles, A.-C. 12

ACID properties 9

    See also Atomicity, Consistency, Durability, Isolation

Action 332–333, 335

ADA 378

ADD 33, 326

Addition rule 84

After-trigger 334

Agent

    See SQL agent

Aggregation 172, 177–178, 181, 213–215, 283–285, 287–288, 540

    See also Average, Count, Data cube, GROUP BY, Maximum, Minimum, Sum

Agrawal, S. 367

Aho, A. V. 122

Algebra 38

    See also Relational algebra

Alias

    See AS

ALL 271, 282–283

ALTER TABLE 33, 326

Ancestor 522

And 254–255

Anomaly 67

    See also Deletion anomaly, Redundancy, Update anomaly

ANSI 243

Antisemijoin 58

ANY 271

Application server 370–372

apply-templates 547

Arithmetic atom 223

Armstrong, W. W. 122

Armstrong’s axioms 81

    See also Augmentation rule, Reflexivity rule, Transitive rule

Array 188, 196, 418

AS 247

Assertion 328–331

Assignment statement 393–394

Association 172–175, 179

Association class 172, 175

Associative array 418

Associative law 212

Astrahan, M. M. 13, 309

Atom 223

Atomicity 2, 9, 298–299

Attribute 22–23, 126–127, 134, 144, 172, 184–185, 194–198, 260, 343, 445, 490–492, 499–502, 506–507, 518, 521

Attribute-based check 320–321, 323, 331

Augmentation rule 81, 83–84

Authorization 425–436

Authorization ID 425

Autoadmin 367–368

Average 214, 284

Axis 517, 521–522

## B

Bag 188–189, 196, 205–212, 228–230

Bancilhon, F. 202, 241

Basis 80

Batini, Carlo 202  
 Batini, Carol 202  
 Battleships database 37, 55–57, 528–529  
**BCNF** 88–92, 111, 113  
 Beeri, C. 122–123  
 Before-trigger 334  
**BEGIN** 394  
 Berenson, H. 309  
 Bernstein, P. A. 123, 309  
 Binary relationship 129–130, 134–135, 172  
 Binding parameters 411  
 Biskup, J. 123  
 Bit string 30, 250  
 Block  
     See Disk block  
 Body 224  
 Boolean 30, 188, 533  
     See also Condition  
 Boyce-Codd normal form  
     See BCNF  
 Bradstock, D. 423  
 Branching 540–541, 551  
     See also ELSE, ELSEIF, IF  
 Bruce, J. 423  
 Buffer manager 7  
 Buneman, P. 515

**C**

C 378  
 Call statement 393, 402  
 Call-level interface 369, 379, 404–405  
     See also CLI  
 Candidate key 72  
 Cartesian product  
     See Product  
 Cascade 314–315  
 Cascade policy 433–436  
 Case sensitivity 248, 530  
 Catalog 373–375  
 Cattell, R. G. G. 202  
**CDATA** 499  
 Celko, J. 309

Ceri, S. 202, 340  
 Chamberlin, D. D. 309, 554  
 Chang, Y.-M. 423  
 Character set 375  
 Character string  
     See String  
 Chase 96–100, 115–119  
 Chaudhuri, S. 367  
**CHECK**  
     See Assertion, Attribute-based check, Tuple-based check  
 Chen, P. P. 202  
 Child 521  
 Choice 505–506  
 Class 172, 179, 184, 188, 193–194, 451  
**CLI** 369, 405–412  
 Client 375–376  
**CLOSE** 385  
 Closed set of attributes 84  
 Closing tag 488  
 Closure, of attributes 75–79  
 Closure, of FD sets 80–81, 115  
 Cluster 374  
 Cobol 378  
 Cochrane, R. J. 340  
**CODASYL** 3  
 Codd, E. F. 3, 65, 123, 241  
 Collation 375  
 Collection type 189  
     See also Array, Bag, Dictionary, List, Set  
 Combining rule 73–74  
 Commit 300  
 Commutative law 212  
 Comparison 461–463, 523–524, 537–538  
     See also Lexicographic order  
 Complementation rule 109–110  
 Complete subclasses 176, 180  
 Complex type 503–506  
 Composition 172, 178, 181  
 Concurrency  
     See Transaction  
 Concurrency control 7–8

Condition 332–334, 523–525  
See also Boolean, Selection, WHERE

Connecting entity set 135, 145

Connection 376–377, 405, 412–413, 419, 427–428

Consistency 9

Constant 38–39

Constraint 18–19, 58–62, 148, 151, 311–331  
See also CHECK, Dependency, Domain constraint, Key, Trigger

Constraint modification 325–327

Containment 59

Correlated subquery 273–274

Count 214, 284–285, 287

CREATE 328–329, 333, 341, 351, 451, 462

CREATE TABLE 30–36, 313, 391, 454

CROSS JOIN 275–276

Cross product  
See Product

Current instance 24

Cursor 383–387, 396, 415, 419–420

**D**

Dangling tuple 219–220, 315

Darwen, H. 309

Data cube 425, 466–467, 473–477

Data model 17–18  
See also Model

Data type  
See UDT

Data warehouse 5, 465

Database 1

Database administrator 5

Database management system  
See DBMS

Database schema 373–375  
See also Relational database schema

Database server 370, 372

Data-definition language 1, 5, 29  
See also ODL, Schema

Datalog 205, 222–238, 439

Data-manipulation language 29

See also Query language

Date 31, 251–252

Date, C. J. 309

Davidson, S. 515

Dayal, U. 123, 340

DBMS 1–10

DDL  
See Data-definition language

Deadlock 9

Decision-support query 464

Declaration 393  
See also CREATE TABLE

DECLARE 381, 397

Decomposition 86–87

Default value 34

Deferrable constraint 316–317

Deferred checking 315–317

Deletion 292–294, 426

Deletion anomaly 86

Delobel, C. 123, 202

Dependency  
See Constraint, Functional dependency, Multivalued dependency

Dependency preservation 93, 100–101, 113

DERIVED 455–456

Descendant 522

Description record 405

Design 140–145, 169  
See also Model, Normalization

Diaz, O. 340

Dicing 469–472

Dictionary 188, 196

Difference 39–40, 50, 207–208, 231, 265–266, 268, 282–283

Dimension table 467–469

Dirty data 302–304

DISCONNECT 377

Disjoint subclasses 176, 180

Disk block 7, 352–353

DISTINCT  
See Duplicate elimination

Distributive law 212–213

DML

See Data-manipulation language Document 488, 499, 502–503, 518–519

Document type definition  
See DTD

DOM 515

Domain 23, 375

Domain constraint 61

Domain relational calculus  
See Relational calculus

Drill-down 471

Driver 412

**DROP** 33, 326, 330, 345

**DROP TABLE** 33

**DTD** 489, 495–502

Duplicate elimination 213–214, 281–284, 538–539

See also **DISTINCT**

Durability 2, 7, 9

Dynamic SQL 388–389

## E

Element 488, 490, 496–497, 503–504, 518

See also Node

Ellis, J. 423

**ELSE** 394

**ELSEIF** 394

Embedded SQL 378–389

Empty element 496

Empty set 59

Empty string 533

**END** 394, 396

Entity 126

Entity set 126–127, 144, 157, 172

See also Connecting entity set, Supporting entity set, Weak entity set

Entity/relationship model

See E/R model

Enumeration 184–185, 188, 508–509

Environment 372–374, 405

Equivalence, of FD's 73

E/R diagram 127–128

E/R model 125–171

Escape character 252

Event 332–334

Event-condition-action rule

See Trigger

**EXCEPT**

See Difference

Exception 400–402

**EXEC SQL** 380

Execute (a SQL statement) 389, 407–408, 413, 419–421, 426

Execution engine 7

**EXISTS** 270

Expression 38, 51

Expression tree 47–48, 236–237

Extended projection 213, 217–219

Extensible markup language

See XML

Extensible modeling language

See XML

Extensible stylesheet language

See XSLT

## F

Fact table 466–467

Fagin, R. 123, 480

Faithfulness 140–141

**FD**

See Functional dependency

FD promotion rule 109

Fetch statement 384, 408–410

Field 509

File system 2

Finkelstein, S. J. 480

First normal form 103

Fisher, M. 423

Floating-point number 31, 188

**FLWR** expression 530–534

For-all 539–540

For-clause 530–533

Foreign key 312–317, 510–512

For-loop 398–400, 549

Fortran 378

**4NF** 110–113

**FROM** 244–246, 259, 274–275

Full outerjoin

See Outerjoin  
Function 391–392, 402  
Functional dependency 67–83  
Functional language 530

## G

Gallaire, H. 241  
Garcia-Molina, H. 65, 515  
Generator 460–461  
Generic interface 245, 378  
Grant diagram 431–432  
Grant statement 375  
Granting privileges 430–431  
Gray, J. N. 309  
Griffiths, P. P. 480  
**GROUP BY** 285–289  
Grouping 213, 215–217, 461  
    See also **GROUP BY**  
Gulutzan, P. 309  
Gupta, A. 241, 367

## H

Handle 405–407  
Harinarayan, V. 241, 367  
**HAVING** 288–289  
Head 224  
Held, G. 13  
Hellerstein, J. M. 13  
Hierarchical model 3, 21  
HiPAC 340  
Host language 245, 369, 378  
Howard, J. H. 123  
HTML 488, 493, 545  
Hull, R. 12

## I

ID 500–502  
    See also Object-ID, Tuple identifier  
IDREF 500–502  
**IF** 394  
Impedance mismatch 380  
**IMPLIED** 499  
**IN** 270–272

**Index** 7–8, 350–358  
Information integration 4–5, 486  
**INGRES** 12  
Inheritance  
    See Isa relationship, Subclass  
Insensitive cursor 388  
**Insert** 461  
Insertion 291–293, 426  
Instance 24, 68, 73, 128–129  
Instead-of-trigger 334, 347–349  
Integer 30, 188  
Interior node 485  
Interpretation of text 417–418, 535–536  
**Intersection** 39–40, 50, 207–208, 212–213, 231, 265, 268, 282–283  
Inverse relationship 186  
Isa relationship 136, 172  
    See also Subclass  
Isolation 2, 9  
Isolation level 304  
    See also Read committed, Read uncommitted, Repeatable read  
Item 518  
Iteration  
    See Loop

## J

**JDBC** 369, 412–416  
**Join** 39, 43, 50, 210–212, 235–236, 259–260, 536–537  
    See also Antisemijoin, CROSS JOIN  
    Lossless join, Natural join, Outerjoin, Semijoin, Theta-join

## K

**Key** 25, 34–36, 60–61, 70, 72, 148–150, 154, 160, 173, 191–192, 311, 353, 509–510  
    See also Foreign key, Primary key, UNIQUE

Kim, W. 202  
Kreps. P. 13

**L**

Label 485  
Leaf 484  
Left outerjoin 221, 277  
Legacy database 486  
Lerdorf, R. 423  
Let-clause 530–531  
Lexicographic order 250  
Ley, M. 12  
Lightstone, S. S. 367  
LIKE 250–251  
Linear recursion 440  
List 188–189, 196  
Liu, M. 241  
Logging 7–8  
Logic

See Datalog, Relational calculus, Three-valued logic

Lohman, G. 367  
Lomet, D. 367  
Loop 396–400, 549  
Lossless join 94–99  
Lowell Report 12

**M**

MacIntyre, P. 423  
Many-many relationship 130–131, 186  
Many-one relationship 129–131, 145, 160, 187  
Materialized view 359–365  
Mattos, N. 340, 480  
Maximum 214, 284  
**maxInclusive** 508  
McCarthy, D. R. 340  
McHugh, J. 515  
Melkanoff, M. A. 123  
Melton, J. 309, 423  
Metadata 8  
    See also Schema  
Method 184, 445, 449, 452–453  
    See also Generator, Mutator

Middleware 5  
Minimal basis 80  
Minimum 214, 284  
**minInclusive** 508  
Minker, J. 241  
Model  
    See E/R model, Hierarchical model, Nested relation, Network model, Object-oriented model, Object-relational model, ODL, Physical data model, Relational model, Semistructured data, UML, XML

Modification 18, 33, 386–387  
    See also Constraint modification, Deletion, Insertion, Updatable view, Update

Module 378  
    See also PSM

**MOLAP** 467  
Monotone operator 57  
Monotonicity 441–443  
Movie database 26–27  
Multidimensional OLAP  
    See MOLAP

Multiset  
    See Bag

Multivalued dependency 67, 105–120  
Multiway relationship 130–131, 134–135, 145

Mumick, I. S. 367, 480  
Mumps 378  
Mutator 460–461  
Mutual recursion 440  
MVD

    See Multivalued dependency

**N**

Nadeau, T. 367  
Namespace 493, 533, 544  
NaN 533  
Narasaya, V. R. 367  
Natural join 43–45, 96, 212, 276–277  
    See also Lossless join

Navathe, S. B. 202  
Negation 254–255  
Nested relation 446–448  
Network model 3, 21  
Nicolas, J.-M. 65  
Node 484, 518–519  
    See also Element  
Nontrivial FD  
    See Trivial FD  
Nontrivial MVD  
    See Trivial MVD  
Normalization 67, 85–92  
Not-null constraint 319–320  
Null value 33–35, 168, 252–254, 287–288, 475  
    See also Not-null constraint, Set-null policy  
Numeric array 418

## O

Object 126, 167–168, 449  
Object description language  
    See ODL  
Object-ID 449, 455–456  
    See also Tuple identifier  
Object-oriented model 21, 449–450  
    See also Object-relational model, ODL  
Object-relational model 20, 445–463  
ODBC  
    See CLI  
ODL 126, 183–198  
OLAP 425, 464–477  
O’Neil, E. 309  
O’Neil, P. 309  
One-one relationship 129–131, 172, 187  
On-line analytic processing  
    See OLAP  
OPEN 384  
Opening tag 488  
Operand 38  
Operator 38  
    See also Monotone operator  
Optimization

    See Query optimization  
Or 254–255  
ORDER BY 255–256, 461  
    See also Ordering, Sorting  
Ordering 461–463, 541–543  
    See also Ordering  
Outerjoin 214, 219–222, 277–278  
Overlapping subclasses 176, 180

## P

Page  
    See Disk block  
Papakonstantinou, Y. 65, 515  
Parameter 391, 410–412, 416  
Parent 522  
Parsed character data  
    See PCDATA  
Partial subclasses 176  
Pascal 378  
Path expression 519–526  
Paton, N. W. 340  
Pattern matching  
    See LIKE  
PCDATA 496  
PEAR 419  
Peer-to-peer system 4  
Pelzer, P. 309  
Persistent stored modules  
    See PSM  
PHP 369, 416–421  
Physical data model 17  
Pirahesh, H. 340, 480  
PL/1 378  
PL/SQL 423  
Predicate 223  
Prepare (a SQL statement) 389, 407, 413, 421  
Prepared statement 413–414  
Preservation of dependencies  
    See Dependency preservation  
Primary key 34–36, 70, 311  
Prime attribute 102  
Privilege 425–436  
Procedure 391–392, 402

Product 39, 43, 50, 210, 235, 259–260  
 Product database 36, 52–54, 526–527  
 Projection 39, 41, 50, 206, 208–209, 232, 246–248  
     See also Extended projection, Lossless join  
 Projection, of FD's 81–83  
 Projection, of MVD's 119–120  
 Prolog 241  
 Proper ancestor 522  
 Proper descendant 522  
 Pseudotransitivity rule 84  
 PSM 391–402  
     See also PL/SQL, SQL PL, Transact-SQL

**Q**

Quantifier  
     See ALL, ANY, EXISTS, For-all, There-exists  
 Quass. D. 241, 515  
 Query 18, 225, 343, 413–414  
     See also Decision-support query  
 Query compiler 7, 10  
 Query language 2  
     See also CLI, Datalog, JDBC, PHP, PSM, Relational algebra, SQL, XPath, XQuery, XSLT  
 Query optimization 18, 49  
 Query optimizer 10  
 Query processing 5, 7, 9–10  
     See also Execution engine, Query compiler  
 Query rewriting 363–364

**R**

Rajaraman, A. 367  
 Ramakrishnan, R. 241  
 Read committed 304–305  
 Read uncommitted 304  
 Read-only transaction 300–302

Real number  
     See Floating-point number  
 Record structure  
     See Structure  
 Recovery 7  
 Recovery of information 93  
     See also Lossless join  
 Recursion 238, 437–443, 546  
 Redundancy 86, 106, 113, 141  
 Reference 446, 449, 454–455, 457–458  
**REFERENCES** 426  
     See also Foreign key  
 Referential integrity 59–60, 150–151, 154, 172, 313–315  
     See also Foreign key  
 Reflexivity rule 81  
 Relation 18, 205, 342  
     See also Table, View  
 Relation instance  
     See Instance  
 Relation schema 22, 24, 29–36  
 Relational algebra 19, 38–52, 59, 205–221, 230–238, 249  
 Relational atom 223  
 Relational calculus 241  
 Relational database schema 22  
 Relational database system 3  
 Relational model 3, 17–19, 21–26, 157–169, 179–183, 193–198, 493–494  
     See also Functional dependency, Multivalued dependency;, Nested relation, Normalization, Object-relational model  
 Relational OLAP  
     See ROLAP  
 Relationship 127, 134, 137, 142–144, 158–160, 185–188, 198  
     See also Binary relationship, Isa relationship, Many-many relationship, Many-one relationship, Multiway relationship, One-one relationship, Supporting relationship

- Relationship set 129  
Relative path expression 521  
Renaming 39, 49–50  
Repeatable read 304–306  
Repeat-loop 399  
**REQUIRED** 499  
**RESTRICT** 433–436  
Return statement 393  
Return-clause 530, 533–534  
Revoking privileges 433–436  
Right outerjoin 221, 277  
ROLAP 467  
Role 131–133, 175  
Rollback 300–301  
Roll-up 471, 476  
Root 485, 489, 495, 519  
Row 22  
    See also Tuple  
Row-level trigger 332, 334  
Rule 224–225  
    See also Safe rule
- S**
- Safe rule 226  
Satisfaction, of an FD 68, 72–73  
SAX 515  
Schema 483–484  
    See also Database schema, Relation schema, Relational database schema, Star schema  
Second normal form 103  
**SELECT** 244–246, 426  
    See also Single-row select  
Selection 39, 42, 50, 209, 232–234, 248–250  
Selection, of indexes 352–358  
Selector 509  
Self 522  
Selinger, P. G.  
    See Griffiths, P. P.  
Semijoin 58  
Semistructured data 18–20, 483–487  
    See also XML  
Sequence 505–506, 518, 535  
Serializability 296–298, 387–388  
Server 375  
    See also Application server, Database server, Web server  
Session 377  
Set 188–189, 195–196, 209, 294, 301, 304, 377, 445  
Set-null policy 314–315  
Shared variable 381–383  
Sibling 522  
Simon, A. R. 309  
Simple type 503, 507–509  
Simplicity 142  
Single-row select 383, 395–396  
Single-value constraint  
    See Functional dependency, Many-one relationship  
Skelley, A. 367  
Slicing 469–472  
SMART 367  
Sorting 214, 219  
    See also ORDER BY, Ordering  
Splitting rule 73–74, 109  
**SQL** 3, 29–36, 243–444, 451–463, 475–477, 530  
SQL agent 378  
SQL PL 423  
SQL state 381, 385  
Star schema 467–469  
Statement 405, 413–415  
Statement-level trigger 332  
Statistics 8  
Stonebraker, M. 13  
Storage manager 7–8  
Stored procedure 375  
    See also PSM  
String 30, 188, 417  
    See also Bit string  
Structure 185, 189, 194–195, 445  
Stylesheet 544  
Subclass 135–138, 165–170, 172, 176, 180–181  
    See also Isa relationship  
Subgoal 224  
Subquery 268–275, 395  
Suciu, D. 515

Sum 214, 284  
 Superkey 71, 88, 102  
 Supporting entity set 154  
 Supporting relationship 154–155  
 Synthesis algorithm for 3NF 103–104  
**SYSTEM GENERATED** 455–456  
 System R 12, 308

**T**

Table 18, 29, 342  
     See also Relation  
 Tableau 97  
 Tag 488, 493  
 Tatroe, K. 423  
 Template 544–548  
 Temporal database 24  
 Temporary table 30  
 Teorey, T. 367  
 Thalheim, B. 202  
 There-exists 539–540  
 Theta-join 45–47  
**3NF** 102–104, 113  
 Three-tier architecture 369–372  
 Three-valued logic 253–255  
 Time 31, 251–252  
 Timestamp 252  
 Transaction 7, 296–306  
 Transact-SQL 423  
 Transitive rule 73, 79–81, 108  
 Tree  
     See Expression tree  
 Trigger 332–337, 426  
 Trivial FD 74–75, 88  
 Trivial MVD 108  
 Truth value 253–255  
 Tuning 357–358, 364–365  
 Tuple 22–23, 449, 458–459  
     See also Dangling tuple  
 Tuple identifier 445–446  
     See also Object-ID  
 Tuple relational calculus  
     See Relational calculus  
 Tuple variable 261–262  
 Tuple-based check 321–323, 331

**Type**  
     See Collection type, Complex type, Data type, Simple type, User-defined type  
 Type constructor 188, 449

**U**

UDT 451–463  
 Ullman, J. D. 13, 122–123, 241, 367, 480  
 UML 125, 171–183  
**UNDER** 426  
 Unicode transformation format  
     See UTF  
 Unified modeling language  
     See UML  
 Union 39–40, 206–207, 212–213, 231, 265–266, 268, 282–283  
**UNIQUE** 34–35, 312  
**UNKNOWN** 253–255  
 Updatable view 345–348  
 Update 294, 413–414, 426  
 Update anomaly 86  
**USAGE** 426  
 User-defined type  
     See UDT  
 UTF 489

**V**

Valentin, G. 367  
 Valid XML 489  
     See also DTD  
**value-of** 545–546  
 Variable 38–39, 223, 232, 417, 534–535  
     See also Tuple variable  
 Variable-length string 30  
 Vianu, V. 12  
 View 29, 341–349  
     See also Materialized view  
 View maintenance 360–362  
 Virtual view  
     See View

**W**

- Wade, B. W. 480  
Weak entity set 152–156, 161–163,  
    181–183  
Web server 370  
Weiner, J. L. 515  
Well-formed XML 489–490  
**WHERE** 244–246, 461  
Where-clause 530, 533  
While-loop 399  
Widom, J. 65, 340, 367, 515  
**WITH** 437  
Wong, E. 13  
World-Wide-Web Consortium 65, 515,  
    554  
W3Schools 515, 554

**X**

- XML 3–4, 19–20, 488–551  
XML Schema 502–512, 523, 533  
XPath 510, 517–526, 530, 545  
XQuery 517, 528, 530–543  
XSLT 517, 544–551

**Z**

- Zaniolo, C. 123  
Zilio, S. 367  
Zuliani, M. 367