Database Management Systems

Second Edition



Raghu Ramakrishnan / Johannes Gehrke

FreeWheel, Ex. 1046

DATABASE MANAGEMENT SYSTEMS

DATABASE MANAGEMENT SYSTEMS

Raghu Ramakrishnan
University of Wisconsin
Madison, WI, USA

Johannes Gehrke
Cornell University
Ithaca, NY, USA



Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Lisbon London Madrid
Mexico City Milan New Delhi Seoul Singapore Sydney Taipei Toronto

McGraw-Hill Higher Education
A Division of the McGraw-Hill Companies

DATABASE MANAGEMENT SYSTEMS

Copyright ©2000, 1998 by The McGraw-Hill Companies, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without prior written permission of the publisher.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 9 0 9 8 7 6 5 4 3 2 1 0 9

ISBN 0-07-232206-3

Vice president/Editor-in-chief: Michael W. Junior
Publisher: Thomas Casson
Executive editor: Elizabeth A. Jones
Developmental editor: Emily J. Gray
Senior marketing manager: John T. Wannemacher
Project manager: Jim Labeots
Senior production supervisor: Heather D. Burbridge
Designer: Kiera Cunningham
Digital control specialist: Brian Nacik
New media: Christopher Styles
Cover illustrator: Mick Wiggins
Typeface: 10/12 Times-Roman
Printer: R.R. Donnelley & Sons Company

Library of Congress Cataloging-in-Publication Data

Ramakrishnan, Raghu.
Database management systems / Raghu Ramakrishnan and Johannes Gehrke —2nd ed.
p. cm.
ISBN 0-07-232206-3
1. Database management. I. Gehrke, Johannes.
II. Title.
QA76.9.D3R237 2000
005.74–dc2d1 99-37109

http://www.mhhe.com

To Apu, Ketan, and Vivek with love

To Keiko

CONTENTS

PREFACE xxii

Part I BASICS 1

1 INTRODUCTION TO DATABASE SYSTEMS 3
1.1 Overview 4
1.2 A Historical Perspective 5
1.3 File Systems versus a DBMS 7
1.4 Advantages of a DBMS 8
1.5 Describing and Storing Data in a DBMS 9
1.5.1 The Relational Model 10
1.5.2 Levels of Abstraction in a DBMS 11
1.5.3 Data Independence 14
1.6 Queries in a DBMS 15
1.7 Transaction Management 15
1.7.1 Concurrent Execution of Transactions 16
1.7.2 Incomplete Transactions and System Crashes 17
1.7.3 Points to Note 18
1.8 Structure of a DBMS 18
1.9 People Who Deal with Databases 20
1.10 Points to Review 21

2 THE ENTITY-RELATIONSHIP MODEL 24
2.1 Overview of Database Design 24
2.1.1 Beyond the ER Model 25
2.2 Entities, Attributes, and Entity Sets 26
2.3 Relationships and Relationship Sets 27
2.4 Additional Features of the ER Model 30
2.4.1 Key Constraints 30
2.4.2 Participation Constraints 32
2.4.3 Weak Entities 33
2.4.4 Class Hierarchies 35
2.4.5 Aggregation 37

2.5 Conceptual Database Design With the ER Model 38
2.5.1 Entity versus Attribute 39
2.5.2 Entity versus Relationship 40
2.5.3 Binary versus Ternary Relationships * 41
2.5.4 Aggregation versus Ternary Relationships * 43
2.6 Conceptual Design for Large Enterprises * 44
2.7 Points to Review 45

3 THE RELATIONAL MODEL 51
3.1 Introduction to the Relational Model 52
3.1.1 Creating and Modifying Relations Using SQL-92 55
3.2 Integrity Constraints over Relations 56
3.2.1 Key Constraints 57
3.2.2 Foreign Key Constraints 59
3.2.3 General Constraints 61
3.3 Enforcing Integrity Constraints 62
3.4 Querying Relational Data 64
3.5 Logical Database Design: ER to Relational 66
3.5.1 Entity Sets to Tables 67
3.5.2 Relationship Sets (without Constraints) to Tables 67
3.5.3 Translating Relationship Sets with Key Constraints 69
3.5.4 Translating Relationship Sets with Participation Constraints 71
3.5.5 Translating Weak Entity Sets 73
3.5.6 Translating Class Hierarchies 74
3.5.7 Translating ER Diagrams with Aggregation 75
3.5.8 ER to Relational: Additional Examples * 76
3.6 Introduction to Views 78
3.6.1 Views, Data Independence, Security 79
3.6.2 Updates on Views 79
3.7 Destroying/Altering Tables and Views 82
3.8 Points to Review 83

Part II RELATIONAL QUERIES 89

4 RELATIONAL ALGEBRA AND CALCULUS 91
4.1 Preliminaries 91
4.2 Relational Algebra 92
4.2.1 Selection and Projection 93
4.2.2 Set Operations 94
4.2.3 Renaming 96
4.2.4 Joins 97
4.2.5 Division 99
4.2.6 More Examples of Relational Algebra Queries 100

4.3 Relational Calculus 106
4.3.1 Tuple Relational Calculus 107
4.3.2 Domain Relational Calculus 111
4.4 Expressive Power of Algebra and Calculus * 114
4.5 Points to Review 115

5 SQL: QUERIES, PROGRAMMING, TRIGGERS 119
5.1 About the Examples 121
5.2 The Form of a Basic SQL Query 121
5.2.1 Examples of Basic SQL Queries 126
5.2.2 Expressions and Strings in the SELECT Command 127
5.3 UNION, INTERSECT, and EXCEPT 129
5.4 Nested Queries 132
5.4.1 Introduction to Nested Queries 132
5.4.2 Correlated Nested Queries 134
5.4.3 Set-Comparison Operators 135
5.4.4 More Examples of Nested Queries 136
5.5 Aggregate Operators 138
5.5.1 The GROUP BY and HAVING Clauses 140
5.5.2 More Examples of Aggregate Queries 143
5.6 Null Values * 147
5.6.1 Comparisons Using Null Values 147
5.6.2 Logical Connectives AND, OR, and NOT 148
5.6.3 Impact on SQL Constructs 148
5.6.4 Outer Joins 149
5.6.5 Disallowing Null Values 150
5.7 Embedded SQL * 150
5.7.1 Declaring Variables and Exceptions 151
5.7.2 Embedding SQL Statements 152
5.8 Cursors * 153
5.8.1 Basic Cursor Definition and Usage 153
5.8.2 Properties of Cursors 155
5.9 Dynamic SQL * 156
5.10 ODBC and JDBC * 157
5.10.1 Architecture 158
5.10.2 An Example Using JDBC 159
5.11 Complex Integrity Constraints in SQL-92 * 161
5.11.1 Constraints over a Single Table 161
5.11.2 Domain Constraints 162
5.11.3 Assertions: ICs over Several Tables 163
5.12 Triggers and Active Databases 164
5.12.1 Examples of Triggers in SQL 165
5.13 Designing Active Databases 166
5.13.1 Why Triggers Can Be Hard to Understand 167

5.13.2 Constraints versus Triggers 167
5.13.3 Other Uses of Triggers 168
5.14 Points to Review 168

6 QUERY-BY-EXAMPLE (QBE) 177
6.1 Introduction 177
6.2 Basic QBE Queries 178
6.2.1 Other Features: Duplicates, Ordering Answers 179
6.3 Queries over Multiple Relations 180
6.4 Negation in the Relation-Name Column 181
6.5 Aggregates 181
6.6 The Conditions Box 183
6.6.1 And/Or Queries 184
6.7 Unnamed Columns 185
6.8 Updates 185
6.8.1 Restrictions on Update Commands 187
6.9 Division and Relational Completeness * 187
6.10 Points to Review 189

Part III DATA STORAGE AND INDEXING 193

7 STORING DATA: DISKS AND FILES 195
7.1 The Memory Hierarchy 196
7.1.1 Magnetic Disks 197
7.1.2 Performance Implications of Disk Structure 199
7.2 RAID 200
7.2.1 Data Striping 200
7.2.2 Redundancy 201
7.2.3 Levels of Redundancy 203
7.2.4 Choice of RAID Levels 206
7.3 Disk Space Management 207
7.3.1 Keeping Track of Free Blocks 207
7.3.2 Using OS File Systems to Manage Disk Space 207
7.4 Buffer Manager 208
7.4.1 Buffer Replacement Policies 211
7.4.2 Buffer Management in DBMS versus OS 212
7.5 Files and Indexes 214
7.5.1 Heap Files 214
7.5.2 Introduction to Indexes 216
7.6 Page Formats * 218
7.6.1 Fixed-Length Records 218
7.6.2 Variable-Length Records 219
7.7 Record Formats * 221

7.7.1 Fixed-Length Records 222
7.7.2 Variable-Length Records 222
7.8 Points to Review 224

8 FILE ORGANIZATIONS AND INDEXES 230
8.1 Cost Model 231
8.2 Comparison of Three File Organizations 232
8.2.1 Heap Files 232
8.2.2 Sorted Files 233
8.2.3 Hashed Files 235
8.2.4 Choosing a File Organization 236
8.3 Overview of Indexes 237
8.3.1 Alternatives for Data Entries in an Index 238
8.4 Properties of Indexes 239
8.4.1 Clustered versus Unclustered Indexes 239
8.4.2 Dense versus Sparse Indexes 241
8.4.3 Primary and Secondary Indexes 242
8.4.4 Indexes Using Composite Search Keys 243
8.5 Index Specification in SQL-92 244
8.6 Points to Review 244

9 TREE-STRUCTURED INDEXING 247
9.1 Indexed Sequential Access Method (ISAM) 248
9.2 B+ Trees: A Dynamic Index Structure 253
9.3 Format of a Node 254
9.4 Search 255
9.5 Insert 257
9.6 Delete * 260
9.7 Duplicates * 265
9.8 B+ Trees in Practice * 266
9.8.1 Key Compression 266
9.8.2 Bulk-Loading a B+ Tree 268
9.8.3 The Order Concept 271
9.8.4 The Effect of Inserts and Deletes on Rids 272
9.9 Points to Review 272

10 HASH-BASED INDEXING 278
10.1 Static Hashing 278
10.1.1 Notation and Conventions 280
10.2 Extendible Hashing * 280
10.3 Linear Hashing * 286
10.4 Extendible Hashing versus Linear Hashing * 291
10.5 Points to Review 292

Part IV QUERY EVALUATION 299

11 EXTERNAL SORTING 301
11.1 A Simple Two-Way Merge Sort 302
11.2 External Merge Sort 305
11.2.1 Minimizing the Number of Runs * 308
11.3 Minimizing I/O Cost versus Number of I/Os 309
11.3.1 Blocked I/O 310
11.3.2 Double Buffering 311
11.4 Using B+ Trees for Sorting 312
11.4.1 Clustered Index 312
11.4.2 Unclustered Index 313
11.5 Points to Review 315

12 EVALUATION OF RELATIONAL OPERATORS 319
12.1 Introduction to Query Processing 320
12.1.1 Access Paths 320
12.1.2 Preliminaries: Examples and Cost Calculations 321
12.2 The Selection Operation 321
12.2.1 No Index, Unsorted Data 322
12.2.2 No Index, Sorted Data 322
12.2.3 B+ Tree Index 323
12.2.4 Hash Index, Equality Selection 324
12.3 General Selection Conditions * 325
12.3.1 CNF and Index Matching 325
12.3.2 Evaluating Selections without Disjunction 326
12.3.3 Selections with Disjunction 327
12.4 The Projection Operation 329
12.4.1 Projection Based on Sorting 329
12.4.2 Projection Based on Hashing * 330
12.4.3 Sorting versus Hashing for Projections * 332
12.4.4 Use of Indexes for Projections * 333
12.5 The Join Operation 333
12.5.1 Nested Loops Join 334
12.5.2 Sort-Merge Join * 339
12.5.3 Hash Join * 343
12.5.4 General Join Conditions * 348
12.6 The Set Operations * 349
12.6.1 Sorting for Union and Difference 349
12.6.2 Hashing for Union and Difference 349
12.7 Aggregate Operations * 350
12.7.1 Implementing Aggregation by Using an Index 351
12.8 The Impact of Buffering * 352

12.9 Points to Review 353

13 INTRODUCTION TO QUERY OPTIMIZATION 359
13.1 Overview of Relational Query Optimization 360
13.1.1 Query Evaluation Plans 361
13.1.2 Pipelined Evaluation 362
13.1.3 The Iterator Interface for Operators and Access Methods 363
13.1.4 The System R Optimizer 364
13.2 System Catalog in a Relational DBMS 365
13.2.1 Information Stored in the System Catalog 365
13.3 Alternative Plans: A Motivating Example 368
13.3.1 Pushing Selections 368
13.3.2 Using Indexes 370
13.4 Points to Review 373

14 A TYPICAL RELATIONAL QUERY OPTIMIZER 374
14.1 Translating SQL Queries into Algebra 375
14.1.1 Decomposition of a Query into Blocks 375
14.1.2 A Query Block as a Relational Algebra Expression 376
14.2 Estimating the Cost of a Plan 378
14.2.1 Estimating Result Sizes 378
14.3 Relational Algebra Equivalences 383
14.3.1 Selections 383
14.3.2 Projections 384
14.3.3 Cross-Products and Joins 384
14.3.4 Selects, Projects, and Joins 385
14.3.5 Other Equivalences 387
14.4 Enumeration of Alternative Plans 387
14.4.1 Single-Relation Queries 387
14.4.2 Multiple-Relation Queries 392
14.5 Nested Subqueries 399
14.6 Other Approaches to Query Optimization 402
14.7 Points to Review 403

Part V DATABASE DESIGN 415

15 SCHEMA REFINEMENT AND NORMAL FORMS 417
15.1 Introduction to Schema Refinement 418
15.1.1 Problems Caused by Redundancy 418
15.1.2 Use of Decompositions 420
15.1.3 Problems Related to Decomposition 421
15.2 Functional Dependencies 422
15.3 Examples Motivating Schema Refinement 423

15.3.1 Constraints on an Entity Set 423
15.3.2 Constraints on a Relationship Set 424
15.3.3 Identifying Attributes of Entities 424
15.3.4 Identifying Entity Sets 426
15.4 Reasoning about Functional Dependencies 427
15.4.1 Closure of a Set of FDs 427
15.4.2 Attribute Closure 429
15.5 Normal Forms 430
15.5.1 Boyce-Codd Normal Form 430
15.5.2 Third Normal Form 432
15.6 Decompositions 434
15.6.1 Lossless-Join Decomposition 435
15.6.2 Dependency-Preserving Decomposition 436
15.7 Normalization 438
15.7.1 Decomposition into BCNF 438
15.7.2 Decomposition into 3NF * 440
15.8 Other Kinds of Dependencies * 444
15.8.1 Multivalued Dependencies 445
15.8.2 Fourth Normal Form 447
15.8.3 Join Dependencies 449
15.8.4 Fifth Normal Form 449
15.8.5 Inclusion Dependencies 449
15.9 Points to Review 450

16 PHYSICAL DATABASE DESIGN AND TUNING 457
16.1 Introduction to Physical Database Design 458
16.1.1 Database Workloads 458
16.1.2 Physical Design and Tuning Decisions 459
16.1.3 Need for Database Tuning 460
16.2 Guidelines for Index Selection 460
16.3 Basic Examples of Index Selection 463
16.4 Clustering and Indexing * 465
16.4.1 Co-clustering Two Relations 468
16.5 Indexes on Multiple-Attribute Search Keys * 470
16.6 Indexes that Enable Index-Only Plans * 471
16.7 Overview of Database Tuning 474
16.7.1 Tuning Indexes 474
16.7.2 Tuning the Conceptual Schema 475
16.7.3 Tuning Queries and Views 476
16.8 Choices in Tuning the Conceptual Schema * 477
16.8.1 Settling for a Weaker Normal Form 478
16.8.2 Denormalization 478
16.8.3 Choice of Decompositions 479
16.8.4 Vertical Decomposition 480

16.8.5 Horizontal Decomposition 481
16.9 Choices in Tuning Queries and Views * 482
16.10 Impact of Concurrency * 484
16.11 DBMS Benchmarking * 485
16.11.1 Well-Known DBMS Benchmarks 486
16.11.2 Using a Benchmark 486
16.12 Points to Review 487

17 SECURITY 497
17.1 Introduction to Database Security 497
17.2 Access Control 498
17.3 Discretionary Access Control 499
17.3.1 Grant and Revoke on Views and Integrity Constraints * 506
17.4 Mandatory Access Control * 508
17.4.1 Multilevel Relations and Polyinstantiation 510
17.4.2 Covert Channels, DoD Security Levels 511
17.5 Additional Issues Related to Security * 512
17.5.1 Role of the Database Administrator 512
17.5.2 Security in Statistical Databases 513
17.5.3 Encryption 514
17.6 Points to Review 517

Part VI TRANSACTION MANAGEMENT 521

18 TRANSACTION MANAGEMENT OVERVIEW 523
18.1 The Concept of a Transaction 523
18.1.1 Consistency and Isolation 525
18.1.2 Atomicity and Durability 525
18.2 Transactions and Schedules 526
18.3 Concurrent Execution of Transactions 527
18.3.1 Motivation for Concurrent Execution 527
18.3.2 Serializability 528
18.3.3 Some Anomalies Associated with Interleaved Execution 528
18.3.4 Schedules Involving Aborted Transactions 531
18.4 Lock-Based Concurrency Control 532
18.4.1 Strict Two-Phase Locking (Strict 2PL) 532
18.5 Introduction to Crash Recovery 533
18.5.1 Stealing Frames and Forcing Pages 535
18.5.2 Recovery-Related Steps during Normal Execution 536
18.5.3 Overview of ARIES 537
18.6 Points to Review 537

19 CONCURRENCY CONTROL 540

19.1 Lock-Based Concurrency Control Revisited 540
19.1.1 2PL, Serializability, and Recoverability 540
19.1.2 View Serializability 543
19.2 Lock Management 543
19.2.1 Implementing Lock and Unlock Requests 544
19.2.2 Deadlocks 546
19.2.3 Performance of Lock-Based Concurrency Control 548
19.3 Specialized Locking Techniques 549
19.3.1 Dynamic Databases and the Phantom Problem 550
19.3.2 Concurrency Control in B+ Trees 551
19.3.3 Multiple-Granularity Locking 554
19.4 Transaction Support in SQL-92 * 555
19.4.1 Transaction Characteristics 556
19.4.2 Transactions and Constraints 558
19.5 Concurrency Control without Locking 559
19.5.1 Optimistic Concurrency Control 559
19.5.2 Timestamp-Based Concurrency Control 561
19.5.3 Multiversion Concurrency Control 563
19.6 Points to Review 564

20 CRASH RECOVERY 571
20.1 Introduction to ARIES 571
20.1.1 The Log 573
20.1.2 Other Recovery-Related Data Structures 576
20.1.3 The Write-Ahead Log Protocol 577
20.1.4 Checkpointing 578
20.2 Recovering from a System Crash 578
20.2.1 Analysis Phase 579
20.2.2 Redo Phase 581
20.2.3 Undo Phase 583
20.3 Media Recovery 586
20.4 Other Algorithms and Interaction with Concurrency Control 587
20.5 Points to Review 588

Part VII ADVANCED TOPICS 595

21 PARALLEL AND DISTRIBUTED DATABASES 597
21.1 Architectures for Parallel Databases 598
21.2 Parallel Query Evaluation 600
21.2.1 Data Partitioning 601
21.2.2 Parallelizing Sequential Operator Evaluation Code 601
21.3 Parallelizing Individual Operations 602
21.3.1 Bulk Loading and Scanning 602

21.3.2 Sorting 602
21.3.3 Joins 603
21.4 Parallel Query Optimization 606
21.5 Introduction to Distributed Databases 607
21.5.1 Types of Distributed Databases 607
21.6 Distributed DBMS Architectures 608
21.6.1 Client-Server Systems 608
21.6.2 Collaborating Server Systems 609
21.6.3 Middleware Systems 609
21.7 Storing Data in a Distributed DBMS 610
21.7.1 Fragmentation 610
21.7.2 Replication 611
21.8 Distributed Catalog Management 611
21.8.1 Naming Objects 612
21.8.2 Catalog Structure 612
21.8.3 Distributed Data Independence 613
21.9 Distributed Query Processing 614
21.9.1 Nonjoin Queries in a Distributed DBMS 614
21.9.2 Joins in a Distributed DBMS 615
21.9.3 Cost-Based Query Optimization 619
21.10 Updating Distributed Data 619
21.10.1 Synchronous Replication 620
21.10.2 Asynchronous Replication 621
21.11 Introduction to Distributed Transactions 624
21.12 Distributed Concurrency Control 625
21.12.1 Distributed Deadlock 625
21.13 Distributed Recovery 627
21.13.1 Normal Execution and Commit Protocols 628
21.13.2 Restart after a Failure 629
21.13.3 Two-Phase Commit Revisited 630
21.13.4 Three-Phase Commit 632
21.14 Points to Review 632

22 INTERNET DATABASES 642
22.1 The World Wide Web 643
22.1.1 Introduction to HTML 643
22.1.2 Databases and the Web 645
22.2 Architecture 645
22.2.1 Application Servers and Server-Side Java 647
22.3 Beyond HTML 651
22.3.1 Introduction to XML 652
22.3.2 XML DTDs 654
22.3.3 Domain-Specific DTDs 657
22.3.4 XML-QL: Querying XML Data 659

22.3.5 The Semistructured Data Model 661
22.3.6 Implementation Issues for Semistructured Data 663
22.4 Indexing for Text Search 663
22.4.1 Inverted Files 665
22.4.2 Signature Files 666
22.5 Ranked Keyword Searches on the Web 667
22.5.1 An Algorithm for Ranking Web Pages 668
22.6 Points to Review 671

23 DECISION SUPPORT 677
23.1 Introduction to Decision Support 678
23.2 Data Warehousing 679
23.2.1 Creating and Maintaining a Warehouse 680
23.3 OLAP 682
23.3.1 Multidimensional Data Model 682
23.3.2 OLAP Queries 685
23.3.3 Database Design for OLAP 689
23.4 Implementation Techniques for OLAP 690
23.4.1 Bitmap Indexes 691
23.4.2 Join Indexes 692
23.4.3 File Organizations 693
23.4.4 Additional OLAP Implementation Issues 693
23.5 Views and Decision Support 694
23.5.1 Views, OLAP, and Warehousing 694
23.5.2 Query Modification 695
23.5.3 View Materialization versus Computing on Demand 696
23.5.4 Issues in View Materialization 698
23.6 Finding Answers Quickly 699
23.6.1 Top N Queries 700
23.6.2 Online Aggregation 701
23.7 Points to Review 702

24 DATA MINING 707
24.1 Introduction to Data Mining 707
24.2 Counting Co-occurrences 708
24.2.1 Frequent Itemsets 709
24.2.2 Iceberg Queries 711
24.3 Mining for Rules 713
24.3.1 Association Rules 714
24.3.2 An Algorithm for Finding Association Rules 714
24.3.3 Association Rules and ISA Hierarchies 715
24.3.4 Generalized Association Rules 716
24.3.5 Sequential Patterns 717

24.3.6 The Use of Association Rules for Prediction 718
24.3.7 Bayesian Networks 719
24.3.8 Classification and Regression Rules 720
24.4 Tree-Structured Rules 722
24.4.1 Decision Trees 723
24.4.2 An Algorithm to Build Decision Trees 725
24.5 Clustering 726
24.5.1 A Clustering Algorithm 728
24.6 Similarity Search over Sequences 729
24.6.1 An Algorithm to Find Similar Sequences 730
24.7 Additional Data Mining Tasks 731
24.8 Points to Review 732

25 OBJECT-DATABASE SYSTEMS 736
25.1 Motivating Example 737
25.1.1 New Data Types 738
25.1.2 Manipulating the New Kinds of Data 739
25.2 User-Defined Abstract Data Types 742
25.2.1 Defining Methods of an ADT 743
25.3 Structured Types 744
25.3.1 Manipulating Data of Structured Types 745
25.4 Objects, Object Identity, and Reference Types 748
25.4.1 Notions of Equality 749
25.4.2 Dereferencing Reference Types 750
25.5 Inheritance 750
25.5.1 Defining Types with Inheritance 751
25.5.2 Binding of Methods 751
25.5.3 Collection Hierarchies, Type Extents, and Queries 752
25.6 Database Design for an ORDBMS 753
25.6.1 Structured Types and ADTs 753
25.6.2 Object Identity 756
25.6.3 Extending the ER Model 757
25.6.4 Using Nested Collections 758
25.7 New Challenges in Implementing an ORDBMS 759
25.7.1 Storage and Access Methods 760
25.7.2 Query Processing 761
25.7.3 Query Optimization 763
25.8 OODBMS 765
25.8.1 The ODMG Data Model and ODL 765
25.8.2 OQL 768
25.9 Comparing RDBMS with OODBMS and ORDBMS 769
25.9.1 RDBMS versus ORDBMS 769
25.9.2 OODBMS versus ORDBMS: Similarities 770
25.9.3 OODBMS versus ORDBMS: Differences 770

25.10 Points to Review 771

26 SPATIAL DATA MANAGEMENT 777
26.1 Types of Spatial Data and Queries 777
26.2 Applications Involving Spatial Data 779
26.3 Introduction to Spatial Indexes 781
26.3.1 Overview of Proposed Index Structures 782
26.4 Indexing Based on Space-Filling Curves 783
26.4.1 Region Quad Trees and Z-Ordering: Region Data 784
26.4.2 Spatial Queries Using Z-Ordering 785
26.5 Grid Files 786
26.5.1 Adapting Grid Files to Handle Regions 789
26.6 R Trees: Point and Region Data 789
26.6.1 Queries 790
26.6.2 Insert and Delete Operations 792
26.6.3 Concurrency Control 793
26.6.4 Generalized Search Trees 794
26.7 Issues in High-Dimensional Indexing 795
26.8 Points to Review 795

27 DEDUCTIVE DATABASES 799
27.1 Introduction to Recursive Queries 800
27.1.1 Datalog 801
27.2 Theoretical Foundations 803
27.2.1 Least Model Semantics 804
27.2.2 Safe Datalog Programs 805
27.2.3 The Fixpoint Operator 806
27.2.4 Least Model = Least Fixpoint 807
27.3 Recursive Queries with Negation 808
27.3.1 Range-Restriction and Negation 809
27.3.2 Stratification 809
27.3.3 Aggregate Operations 812
27.4 Efficient Evaluation of Recursive Queries 813
27.4.1 Fixpoint Evaluation without Repeated Inferences 814
27.4.2 Pushing Selections to Avoid Irrelevant Inferences 816
27.5 Points to Review 818

28 ADDITIONAL TOPICS 822
28.1 Advanced Transaction Processing 822
28.1.1 Transaction Processing Monitors 822
28.1.2 New Transaction Models 823
28.1.3 Real-Time DBMSs 824
28.2 Integrated Access to Multiple Data Sources 824

28.3 Mobile Databases 825
28.4 Main Memory Databases 825
28.5 Multimedia Databases 826
28.6 Geographic Information Systems 827
28.7 Temporal and Sequence Databases 828
28.8 Information Visualization 829
28.9 Summary 829

A DATABASE DESIGN CASE STUDY: THE INTERNET SHOP 831
A.1 Requirements Analysis 831
A.2 Conceptual Design 832
A.3 Logical Database Design 832
A.4 Schema Refinement 835
A.5 Physical Database Design 836
A.5.1 Tuning the Database 838
A.6 Security 838
A.7 Application Layers 840

B THE MINIBASE SOFTWARE 842
B.1 What's Available 842
B.2 Overview of Minibase Assignments 843
B.2.1 Overview of Programming Projects 843
B.2.2 Overview of Nonprogramming Assignments 844
B.3 Acknowledgments 845

REFERENCES 847

SUBJECT INDEX 879

AUTHOR INDEX 896

PREFACE

> The advantage of doing one's praising for oneself is that one can lay it on so thick and exactly in the right places.
>
> —Samuel Butler

Database management systems have become ubiquitous as a fundamental tool for managing information, and a course on the principles and practice of database systems is now an integral part of computer science curricula. This book covers the fundamentals of modern database management systems, in particular relational database systems. It is intended as a text for an introductory database course for undergraduates, and we have attempted to present the material in a clear, simple style.

A quantitative approach is used throughout and detailed examples abound. An extensive set of exercises (for which solutions are available online to instructors) accompanies each chapter and reinforces students' ability to apply the concepts to real problems. The book contains enough material to support a second course, ideally supplemented by selected research papers. It can be used, with the accompanying software and SQL programming assignments, in two distinct kinds of introductory courses:

1. A course that aims to present the principles of database systems, with a practical focus but without any implementation assignments. The SQL programming assignments are a useful supplement for such a course. The supplementary Minibase software can be used to create exercises and experiments with no programming.

2. A course that has a strong systems emphasis and assumes that students have good programming skills in C and C++. In this case the software can be used as the basis for projects in which students are asked to implement various parts of a relational DBMS. Several central modules in the project software (e.g., heap files, buffer manager, B+ trees, hash indexes, various join methods, concurrency control, and recovery algorithms) are described in sufficient detail in the text to enable students to implement them, given the (C++) class interfaces.

Many instructors will no doubt teach a course that falls between these two extremes.

Choice of Topics

The choice of material has been influenced by these considerations:

- To concentrate on issues central to the *design, tuning, and implementation of relational database applications*. However, many of the issues discussed (e.g., buffering and access methods) are not specific to relational systems, and additional topics such as decision support and object-database systems are covered in later chapters.
- To provide adequate coverage of implementation topics to support a concurrent laboratory section or course project. For example, implementation of relational operations has been covered in more detail than is necessary in a first course. However, the variety of alternative implementation techniques permits a wide choice of project assignments. An instructor who wishes to assign implementation of sort-merge join might cover that topic in depth, whereas another might choose to emphasize index nested loops join.
- To provide in-depth coverage of the state of the art in currently available commercial systems, rather than a broad coverage of several alternatives. For example, we discuss the relational data model, B+ trees, SQL, System R style query optimization, lock-based concurrency control, the ARIES recovery algorithm, the two-phase commit protocol, asynchronous replication in distributed databases, and object-relational DBMSs in detail, with numerous illustrative examples. This is made possible by omitting or briefly covering some related topics such as the hierarchical and network models, B tree variants, Quel, semantic query optimization, view serializability, the shadow-page recovery algorithm, and the three-phase commit protocol.
- The same preference for in-depth coverage of selected topics governed our choice of topics for chapters on advanced material. Instead of covering a broad range of topics briefly, we have chosen topics that we believe to be practically important and at the cutting edge of current thinking in database systems, and we have covered them in depth.

New in the Second Edition

Based on extensive user surveys and feedback, we have refined the book's organization. The major change is the early introduction of the ER model, together with a discussion of conceptual database design. As in the first edition, we introduce SQL-92's data definition features together with the relational model (in Chapter 3), and whenever appropriate, relational model concepts (e.g., definition of a relation, updates, views, ER to relational mapping) are illustrated and discussed in the context of SQL. Of course, we maintain a careful separation between the concepts and their SQL realization. The material on data storage, file organization, and indexes has been moved back, and the

material on relational queries has been moved forward. Nonetheless, the two parts (storage and organization vs. queries) can still be taught in either order based on the instructor's preferences.

In order to facilitate brief coverage in a first course, the second edition contains overview chapters on transaction processing and query optimization. Most chapters have been revised extensively, and additional explanations and figures have been added in many places. For example, the chapters on query languages now contain a uniform numbering of all queries to facilitate comparisons of the same query (in algebra, calculus, and SQL), and the results of several queries are shown in figures. JDBC and ODBC coverage has been added to the SQL query chapter and SQL:1999 features are discussed both in this chapter and the chapter on object-relational databases. A discussion of RAID has been added to Chapter 7. We have added a new database design case study, illustrating the entire design cycle, as an appendix.

Two new pedagogical features have been introduced. First, 'floating boxes' provide additional perspective and relate the concepts to real systems, while keeping the main discussion free of product-specific details. Second, each chapter concludes with a 'Points to Review' section that summarizes the main ideas introduced in the chapter and includes pointers to the sections where they are discussed.

For use in a second course, many advanced chapters from the first edition have been extended or split into multiple chapters to provide thorough coverage of current topics. In particular, new material has been added to the chapters on decision support, deductive databases, and object databases. New chapters on Internet databases, data mining, and spatial databases have been added, greatly expanding the coverage of these topics.

The material can be divided into roughly seven parts, as indicated in Figure 0.1, which also shows the dependencies between chapters. An arrow from Chapter I to Chapter J means that I depends on material in J. The broken arrows indicate a weak dependency, which can be ignored at the instructor's discretion. It is recommended that Part I be covered first, followed by Part II and Part III (in either order). Other than these three parts, dependencies across parts are minimal.

Order of Presentation

The book's modular organization offers instructors a variety of choices. For example, some instructors will want to cover SQL and get students to use a relational database, before discussing file organizations or indexing; they should cover Part II before Part III. In fact, in a course that emphasizes concepts and SQL, many of the implementation-oriented chapters might be skipped. On the other hand, instructors assigning implementation projects based on file organizations may want to cover Part

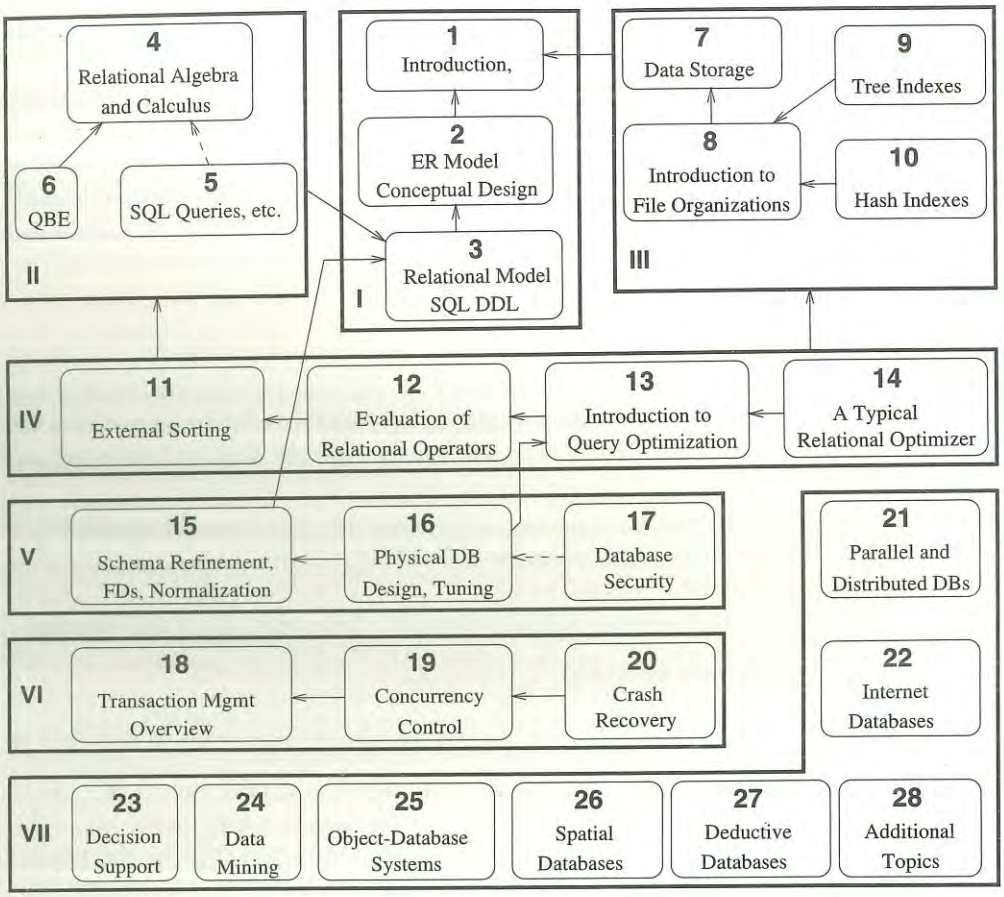


Figure 0.1 Chapter Organization and Dependencies

III early to space assignments. As another example, it is not necessary to cover all the alternatives for a given operator (e.g., various techniques for joins) in Chapter 12 in order to cover later related material (e.g., on optimization or tuning) adequately. The database design case study in the appendix can be discussed concurrently with the appropriate design chapters, or it can be discussed after all design topics have been covered, as a review.

Several section headings contain an asterisk. This symbol does not necessarily indicate a higher level of difficulty. Rather, omitting all asterisked sections leaves about the right amount of material in Chapters 1–18, possibly omitting Chapters 6, 10, and 14, for a broad introductory one-quarter or one-semester course (depending on the depth at which the remaining material is discussed and the nature of the course assignments).

The book can be used in several kinds of introductory or second courses by choosing topics appropriately, or in a two-course sequence by supplementing the material with some advanced readings in the second course. Examples of appropriate introductory courses include courses on file organizations and introduction to database management systems, especially if the course focuses on relational database design or implementation. Advanced courses can be built around the later chapters, which contain detailed bibliographies with ample pointers for further study.

Supplementary Material

Each chapter contains several exercises designed to test and expand the reader's understanding of the material. Students can obtain solutions to odd-numbered chapter exercises and a set of lecture slides for each chapter through the Web in Postscript and Adobe PDF formats.

The following material is available online to instructors:

1. Lecture slides for all chapters in MS Powerpoint, Postscript, and PDF formats.
2. Solutions to all chapter exercises.
3. SQL queries and programming assignments with solutions. (This is new for the second edition.)
4. Supplementary project software (Minibase) with sample assignments and solutions, as described in Appendix B. The text itself does not refer to the project software, however, and can be used independently in a course that presents the principles of database management systems from a practical perspective, but without a project component.

The supplementary material on SQL is new for the second edition. The remaining material has been extensively revised from the first edition versions.

For More Information

The home page for this book is at URL:

`http://www.cs.wisc.edu/~dbbook`

This page is frequently updated *and contains a link to all known errors in the book, the accompanying slides, and the supplements.* Instructors should visit this site periodically or register at this site to be notified of important changes by email.

Acknowledgments

This book grew out of lecture notes for CS564, the introductory (senior/graduate level) database course at UW-Madison. David DeWitt developed this course and the Minirel project, in which students wrote several well-chosen parts of a relational DBMS. My thinking about this material was shaped by teaching CS564, and Minirel was the inspiration for Minibase, which is more comprehensive (e.g., it has a query optimizer and includes visualization software) but tries to retain the spirit of Minirel. Mike Carey and I jointly designed much of Minibase. My lecture notes (and in turn this book) were influenced by Mike's lecture notes and by Yannis Ioannidis's lecture slides.

Joe Hellerstein used the beta edition of the book at Berkeley and provided invaluable feedback, assistance on slides, and hilarious quotes. Writing the chapter on object-database systems with Joe was a lot of fun.

C. Mohan provided invaluable assistance, patiently answering a number of questions about implementation techniques used in various commercial systems, in particular indexing, concurrency control, and recovery algorithms. Moshe Zloof answered numerous questions about QBE semantics and commercial systems based on QBE. Ron Fagin, Krishna Kulkarni, Len Shapiro, Jim Melton, Dennis Shasha, and Dirk Van Gucht reviewed the book and provided detailed feedback, greatly improving the content and presentation. Michael Goldweber at Beloit College, Matthew Haines at Wyoming, Michael Kifer at SUNY StonyBrook, Jeff Naughton at Wisconsin, Praveen Seshadri at Cornell, and Stan Zdonik at Brown also used the beta edition in their database courses and offered feedback and bug reports. In particular, Michael Kifer pointed out an error in the (old) algorithm for computing a minimal cover and suggested covering some SQL features in Chapter 2 to improve modularity. Gio Wiederhold's bibliography, converted to Latex format by S. Sudarshan, and Michael Ley's online bibliography on databases and logic programming were a great help while compiling the chapter bibliographies. Shaun Flisakowski and Uri Shaft helped me frequently in my never-ending battles with Latex.

I owe a special thanks to the many, many students who have contributed to the Minibase software. Emmanuel Ackaouy, Jim Pruyne, Lee Schumacher, and Michael Lee worked with me when I developed the first version of Minibase (much of which was subsequently discarded, but which influenced the next version). Emmanuel Ackaouy and Bryan So were my TAs when I taught CS564 using this version and went well beyond the limits of a TAship in their efforts to refine the project. Paul Aoki struggled with a version of Minibase and offered lots of useful comments as a TA at Berkeley. An entire class of CS764 students (our graduate database course) developed much of the current version of Minibase in a large class project that was led and coordinated by Mike Carey and me. Amit Shukla and Michael Lee were my TAs when I first taught CS564 using this version of Minibase and developed the software further.

Several students worked with me on independent projects, over a long period of time, to develop Minibase components. These include visualization packages for the buffer manager and B+ trees (Huseyin Bektas, Harry Stavropoulos, and Weiqing Huang); a query optimizer and visualizer (Stephen Harris, Michael Lee, and Donko Donjerkovic); an ER diagram tool based on the Opossum schema editor (Eben Haber); and a GUI-based tool for normalization (Andrew Prock and Andy Therber). In addition, Bill Kimmel worked to integrate and fix a large body of code (storage manager, buffer manager, files and access methods, relational operators, and the query plan executor) produced by the CS764 class project. Ranjani Ramamurty considerably extended Bill's work on cleaning up and integrating the various modules. Luke Blanshard, Uri Shaft, and Shaun Flisakowski worked on putting together the release version of the code and developed test suites and exercises based on the Minibase software. Krishna Kunchithapadam tested the optimizer and developed part of the Minibase GUI.

Clearly, the Minibase software would not exist without the contributions of a great many talented people. With this software available freely in the public domain, I hope that more instructors will be able to teach a systems-oriented database course with a blend of implementation and experimentation to complement the lecture material.

I'd like to thank the many students who helped in developing and checking the solutions to the exercises and provided useful feedback on draft versions of the book. In alphabetical order: X. Bao, S. Biao, M. Chakrabarti, C. Chan, W. Chen, N. Cheung, D. Colwell, C. Fritz, V. Ganti, J. Gehrke, G. Glass, V. Gopalakrishnan, M. Higgins, T. Jasmin, M. Krishnaprasad, Y. Lin, C. Liu, M. Lusignan, H. Modi, S. Narayanan, D. Randolph, A. Ranganathan, J. Reminga, A. Therber, M. Thomas, Q. Wang, R. Wang, Z. Wang, and J. Yuan. Arcady Grenader, James Harrington, and Martin Reames at Wisconsin and Nina Tang at Berkeley provided especially detailed feedback.

Charlie Fischer, Avi Silberschatz, and Jeff Ullman gave me invaluable advice on working with a publisher. My editors at McGraw-Hill, Betsy Jones and Eric Munson, obtained extensive reviews and guided this book in its early stages. Emily Gray and Brad Kosirog were there whenever problems cropped up. At Wisconsin, Ginny Werner really helped me to stay on top of things.

Finally, this book was a thief of time, and in many ways it was harder on my family than on me. My sons expressed themselves forthrightly. From my (then) five-year-old, Ketan: "Dad, stop working on that silly book. You don't have any time for *me*." Two-year-old Vivek: "You working *boook*? No no no come play basketball me!" All the seasons of their discontent were visited upon my wife, and Apu nonetheless cheerfully kept the family going in its usual chaotic, happy way all the many evenings and weekends I was wrapped up in this book. (Not to mention the days when I was wrapped up in being a faculty member!) As in all things, I can trace my parents' hand in much of this; my father, with his love of learning, and my mother, with her love of us, shaped me. My brother Kartik's contributions to this book consisted chiefly of

phone calls in which he kept me from working, but if I don't acknowledge him, he's liable to be annoyed. I'd like to thank my family for being there and giving meaning to everything I do. (There! I knew I'd find a legitimate reason to thank Kartik.)

Acknowledgments for the Second Edition

Emily Gray and Betsy Jones at McGraw-Hill obtained extensive reviews and provided guidance and support as we prepared the second edition. Jonathan Goldstein helped with the bibliography for spatial databases. The following reviewers provided valuable feedback on content and organization: Liming Cai at Ohio University, Costas Tsatsoulis at University of Kansas, Kwok-Bun Yue at University of Houston, Clear Lake, William Grosky at Wayne State University, Sang H. Son at University of Virginia, James M. Slack at Minnesota State University, Mankato, Herman Balsters at University of Twente, Netherlands, Karen C. Davis at University of Cincinnati, Joachim Hammer at University of Florida, Fred Petry at Tulane University, Gregory Speegle at Baylor University, Salih Yurttas at Texas A&M University, and David Chao at San Francisco State University.

A number of people reported bugs in the first edition. In particular, we wish to thank the following: Joseph Albert at Portland State University, Han-yin Chen at University of Wisconsin, Lois Delcambre at Oregon Graduate Institute, Maggie Eich at Southern Methodist University, Raj Gopalan at Curtin University of Technology, Davood Rafiei at University of Toronto, Michael Schrefl at University of South Australia, Alex Thomasian at University of Connecticut, and Scott Vandenberg at Siena College.

A special thanks to the many people who answered a detailed survey about how commercial systems support various features: At IBM, Mike Carey, Bruce Lindsay, C. Mohan, and James Teng; at Informix, M. Muralikrishna and Michael Ubell; at Microsoft, David Campbell, Goetz Graefe, and Peter Spiro; at Oracle, Hakan Jacobsson, Jonathan D. Klein, Muralidhar Krishnaprasad, and M. Ziauddin; and at Sybase, Marc Chanliau, Lucien Dimino, Sangeeta Doraiswamy, Hanuma Kodavalla, Roger MacNicol, and Tirumanjanam Rengarajan.

After reading about himself in the acknowledgment to the first edition, Ketan (now 8) had a simple question: "How come you didn't dedicate the book to us? Why mom?" Ketan, I took care of this inexplicable oversight. Vivek (now 5) was more concerned about the extent of his fame: "Daddy, is my name in *evvy* copy of your book? Do they have it in *evvy* compooter science department in the world?" Vivek, I hope so. Finally, this revision would not have made it without Apu's and Keiko's support.

PART I

BASICS

1

INTRODUCTION TO DATABASE SYSTEMS

> Has everyone noticed that all the letters of the word *database* are typed with the left hand? Now the layout of the QWERTY typewriter keyboard was designed, among other things, to facilitate the even use of both hands. It follows, therefore, that writing about databases is not only unnatural, but a lot harder than it appears.
>
> —Anonymous

Today, more than at any previous time, the success of an organization depends on its ability to acquire accurate and timely data about its operations, to manage this data effectively, and to use it to analyze and guide its activities. Phrases such as the *information superhighway* have become ubiquitous, and information processing is a rapidly growing multibillion dollar industry.

The amount of information available to us is literally exploding, and the value of data as an organizational asset is widely recognized. Yet without the ability to manage this vast amount of data, and to quickly find the information that is relevant to a given question, as the amount of information increases, it tends to become a distraction and a liability, rather than an asset. This paradox drives the need for increasingly powerful and flexible data management systems. To get the most out of their large and complex datasets, users must have tools that simplify the tasks of managing the data and extracting useful information in a timely fashion. Otherwise, data can become a liability, with the cost of acquiring it and managing it far exceeding the value that is derived from it.

A **database** is a collection of data, typically describing the activities of one or more related organizations. For example, a university database might contain information about the following:

Entities such as students, faculty, courses, and classrooms.

Relationships between entities, such as students' enrollment in courses, faculty teaching courses, and the use of rooms for courses.

A **database management system**, or **DBMS**, is software designed to assist in maintaining and utilizing large collections of data, and the need for such systems, as well as their use, is growing rapidly. The alternative to using a DBMS is to use ad

hoc approaches that do not carry over from one application to another; for example, to store the data in files and write application-specific code to manage it. The use of a DBMS has several important advantages, as we will see in Section 1.4.

The area of database management systems is a microcosm of computer science in general. The issues addressed and the techniques used span a wide spectrum, including languages, object-orientation and other programming paradigms, compilation, operating systems, concurrent programming, data structures, algorithms, theory, parallel and distributed systems, user interfaces, expert systems and artificial intelligence, statistical techniques, and dynamic programming. We will not be able to go into all these aspects of database management in this book, but it should be clear that this is a rich and vibrant discipline.

1.1 OVERVIEW

The goal of this book is to present an in-depth introduction to database management systems, with an emphasis on how to organize information in a DBMS and to maintain it and retrieve it efficiently, that is, how to *design* a database and *use* a DBMS effectively. Not surprisingly, many decisions about how to use a DBMS for a given application depend on what capabilities the DBMS supports efficiently. Thus, to use a DBMS well, it is necessary to also understand how a DBMS *works*. The approach taken in this book is to emphasize how to *use* a DBMS, while covering DBMS implementation and architecture in sufficient detail to understand how to *design a database*.

Many kinds of database management systems are in use, but this book concentrates on *relational* systems, which are by far the dominant type of DBMS today. The following questions are addressed in the core chapters of this book:

1. **Database Design:** How can a user describe a real-world enterprise (e.g., a university) in terms of the data stored in a DBMS? What factors must be considered in deciding how to organize the stored data? (Chapters 2, 3, 15, 16, and 17.)

2. **Data Analysis:** How can a user answer questions about the enterprise by posing queries over the data in the DBMS? (Chapters 4, 5, 6, and 23.)

3. **Concurrency and Robustness:** How does a DBMS allow many users to access data concurrently, and how does it protect the data in the event of system failures? (Chapters 18, 19, and 20.)

4. **Efficiency and Scalability:** How does a DBMS store large datasets and answer questions against this data efficiently? (Chapters 7, 8, 9, 10, 11, 12, 13, and 14.)

Later chapters cover important and rapidly evolving topics such as parallel and distributed database management, Internet databases, data warehousing and complex

queries for decision support, data mining, object databases, spatial data management, and rule-oriented DBMS extensions.

In the rest of this chapter, we introduce the issues listed above. In Section 1.2, we begin with a brief history of the field and a discussion of the role of database management in modern information systems. We then identify benefits of storing data in a DBMS instead of a file system in Section 1.3, and discuss the advantages of using a DBMS to manage data in Section 1.4. In Section 1.5 we consider how information about an enterprise should be organized and stored in a DBMS. A user probably thinks about this information in high-level terms corresponding to the entities in the organization and their relationships, whereas the DBMS ultimately stores data in the form of (many, many) bits. The gap between how users think of their data and how the data is ultimately stored is bridged through several *levels of abstraction* supported by the DBMS. Intuitively, a user can begin by describing the data in fairly high-level terms, and then refine this description by considering additional storage and representation details as needed.

In Section 1.6 we consider how users can retrieve data stored in a DBMS and the need for techniques to efficiently compute answers to questions involving such data. In Section 1.7 we provide an overview of how a DBMS supports concurrent access to data by several users, and how it protects the data in the event of system failures.

We then briefly describe the internal structure of a DBMS in Section 1.8, and mention various groups of people associated with the development and use of a DBMS in Section 1.9.

1.2 A HISTORICAL PERSPECTIVE

From the earliest days of computers, storing and manipulating data have been a major application focus. The first general-purpose DBMS was designed by Charles Bachman at General Electric in the early 1960s and was called the Integrated Data Store. It formed the basis for the *network data model*, which was standardized by the Conference on Data Systems Languages (CODASYL) and strongly influenced database systems through the 1960s. Bachman was the first recipient of ACM's Turing Award (the computer science equivalent of a Nobel prize) for work in the database area; he received the award in 1973.

In the late 1960s, IBM developed the Information Management System (IMS) DBMS, used even today in many major installations. IMS formed the basis for an alternative data representation framework called the *hierarchical data model*. The SABRE system for making airline reservations was jointly developed by American Airlines and IBM around the same time, and it allowed several people to access the same data through

a computer network. Interestingly, today the same SABRE system is used to power popular Web-based travel services such as Travelocity!

In 1970, Edgar Codd, at IBM's San Jose Research Laboratory, proposed a new data representation framework called the *relational data model*. This proved to be a watershed in the development of database systems: it sparked rapid development of several DBMSs based on the relational model, along with a rich body of theoretical results that placed the field on a firm foundation. Codd won the 1981 Turing Award for his seminal work. Database systems matured as an academic discipline, and the popularity of relational DBMSs changed the commercial landscape. Their benefits were widely recognized, and the use of DBMSs for managing corporate data became standard practice.

In the 1980s, the relational model consolidated its position as the dominant DBMS paradigm, and database systems continued to gain widespread use. The SQL query language for relational databases, developed as part of IBM's System R project, is now the standard query language. SQL was standardized in the late 1980s, and the current standard, SQL-92, was adopted by the American National Standards Institute (ANSI) and International Standards Organization (ISO). Arguably, the most widely used form of concurrent programming is the concurrent execution of database programs (called *transactions*). Users write programs as if they are to be run by themselves, and the responsibility for running them concurrently is given to the DBMS. James Gray won the 1999 Turing award for his contributions to the field of transaction management in a DBMS.

In the late 1980s and the 1990s, advances have been made in many areas of database systems. Considerable research has been carried out into more powerful query languages and richer data models, and there has been a big emphasis on supporting complex analysis of data from all parts of an enterprise. Several vendors (e.g., IBM's DB2, Oracle 8, Informix UDS) have extended their systems with the ability to store new data types such as images and text, and with the ability to ask more complex queries. Specialized systems have been developed by numerous vendors for creating *data warehouses*, consolidating data from several databases, and for carrying out specialized analysis.

An interesting phenomenon is the emergence of several *enterprise resource planning (ERP)* and *management resource planning (MRP)* packages, which add a substantial layer of application-oriented features on top of a DBMS. Widely used packages include systems from Baan, Oracle, PeopleSoft, SAP, and Siebel. These packages identify a set of common tasks (e.g., inventory management, human resources planning, financial analysis) encountered by a large number of organizations and provide a general application layer to carry out these tasks. The data is stored in a relational DBMS, and the application layer can be customized to different companies, leading to lower

overall costs for the companies, compared to the cost of building the application layer from scratch.

Most significantly, perhaps, DBMSs have entered the Internet Age. While the first generation of Web sites stored their data exclusively in operating systems files, the use of a DBMS to store data that is accessed through a Web browser is becoming widespread. Queries are generated through Web-accessible forms and answers are formatted using a markup language such as HTML, in order to be easily displayed in a browser. All the database vendors are adding features to their DBMS aimed at making it more suitable for deployment over the Internet.

Database management continues to gain importance as more and more data is brought on-line, and made ever more accessible through computer networking. Today the field is being driven by exciting visions such as multimedia databases, interactive video, digital libraries, a host of scientific projects such as the human genome mapping effort and NASA's Earth Observation System project, and the desire of companies to consolidate their decision-making processes and *mine* their data repositories for useful information about their businesses. Commercially, database management systems represent one of the largest and most vigorous market segments. Thus the study of database systems could prove to be richly rewarding in more ways than one!

1.3 FILE SYSTEMS VERSUS A DBMS

To understand the need for a DBMS, let us consider a motivating scenario: A company has a large collection (say, 500 GB[1]) of data on employees, departments, products, sales, and so on. This data is accessed concurrently by several employees. Questions about the data must be answered quickly, changes made to the data by different users must be applied consistently, and access to certain parts of the data (e.g., salaries) must be restricted.

We can try to deal with this data management problem by storing the data in a collection of operating system files. This approach has many drawbacks, including the following:

- We probably do not have 500 GB of main memory to hold all the data. We must therefore store data in a storage device such as a disk or tape and bring relevant parts into main memory for processing as needed.
- Even if we have 500 GB of main memory, on computer systems with 32-bit addressing, we cannot refer directly to more than about 4 GB of data! We have to program some method of identifying all data items.

[1]A kilobyte (KB) is 1024 bytes, a megabyte (MB) is 1024 KBs, a gigabyte (GB) is 1024 MBs, a terabyte (TB) is 1024 GBs, and a petabyte (PB) is 1024 terabytes.

- We have to write special programs to answer each question that users may want to ask about the data. These programs are likely to be complex because of the large volume of data to be searched.
- We must protect the data from inconsistent changes made by different users accessing the data concurrently. If programs that access the data are written with such concurrent access in mind, this adds greatly to their complexity.
- We must ensure that data is restored to a consistent state if the system crashes while changes are being made.
- Operating systems provide only a password mechanism for security. This is not sufficiently flexible to enforce security policies in which different users have permission to access different subsets of the data.

A DBMS is a piece of software that is designed to make the preceding tasks easier. By storing data in a DBMS, rather than as a collection of operating system files, we can use the DBMS's features to manage the data in a robust and efficient manner. As the volume of data and the number of users grow—hundreds of gigabytes of data and thousands of users are common in current corporate databases—DBMS support becomes indispensable.

1.4 ADVANTAGES OF A DBMS

Using a DBMS to manage data has many advantages:

- **Data independence:** Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.
- **Efficient data access:** A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.
- **Data integrity and security:** If data is always accessed through the DBMS, the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce *access controls* that govern what data is visible to different classes of users.
- **Data administration:** When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and for fine-tuning the storage of the data to make retrieval efficient.

- **Concurrent access and crash recovery:** A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.
- **Reduced application development time:** Clearly, the DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick development of applications. Such applications are also likely to be more robust than applications developed from scratch because many important tasks are handled by the DBMS instead of being implemented by the application.

Given all these advantages, is there ever a reason *not* to use a DBMS? A DBMS is a complex piece of software, optimized for certain kinds of workloads (e.g., answering complex queries or handling many concurrent requests), and its performance may not be adequate for certain specialized applications. Examples include applications with tight real-time constraints or applications with just a few well-defined critical operations for which efficient custom code must be written. Another reason for not using a DBMS is that an application may need to manipulate the data in ways not supported by the query language. In such a situation, the abstract view of the data presented by the DBMS does not match the application's needs, and actually gets in the way. As an example, relational databases do not support flexible analysis of text data (although vendors are now extending their products in this direction). If specialized performance or data manipulation requirements are central to an application, the application may choose not to use a DBMS, especially if the added benefits of a DBMS (e.g., flexible querying, security, concurrent access, and crash recovery) are not required. In most situations calling for large-scale data management, however, DBMSs have become an indispensable tool.

1.5 DESCRIBING AND STORING DATA IN A DBMS

The user of a DBMS is ultimately concerned with some real-world enterprise, and the data to be stored describes various aspects of this enterprise. For example, there are students, faculty, and courses in a university, and the data in a university database describes these entities and their relationships.

A **data model** is a collection of high-level data description constructs that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model. Most database management systems today are based on the **relational data model**, which we will focus on in this book.

While the data model of the DBMS hides many details, it is nonetheless closer to how the DBMS stores data than to how a user thinks about the underlying enterprise. A **semantic data model** is a more abstract, high-level data model that makes it easier

for a user to come up with a good initial description of the data in an enterprise. These models contain a wide variety of constructs that help describe a real application scenario. A DBMS is not intended to support all these constructs directly; it is typically built around a data model with just a few basic constructs, such as the relational model. A database design in terms of a semantic model serves as a useful starting point and is subsequently translated into a database design in terms of the data model the DBMS actally supports.

A widely used semantic data model called the entity-relationship (ER) model allows us to pictorially denote entities and the relationships among them. We cover the ER model in Chapter 2.

1.5.1 The Relational Model

In this section we provide a brief introduction to the relational model. The central data description construct in this model is a **relation**, which can be thought of as a set of **records**.

A description of data in terms of a data model is called a **schema**. In the relational model, the schema for a relation specifies its name, the name of each **field** (or **attribute** or **column**), and the type of each field. As an example, student information in a university database may be stored in a relation with the following schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`, *age:* `integer`, *gpa:* `real`)

The preceding schema says that each record in the Students relation has five fields, with field names and types as indicated.[2] An example instance of the Students relation appears in Figure 1.1.

sid	*name*	*login*	*age*	*gpa*
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 1.1 An Instance of the Students Relation

[2]Storing *date of birth* is preferable to storing *age*, since it does not change over time, unlike age. We've used *age* for simplicity in our discussion.

Each row in the Students relation is a record that describes a student. The description is not complete—for example, the student's height is not included—but is presumably adequate for the intended applications in the university database. Every row follows the schema of the Students relation. The schema can therefore be regarded as a template for describing a student.

We can make the description of a collection of students more precise by specifying **integrity constraints**, which are conditions that the records in a relation must satisfy. For example, we could specify that every student has a unique *sid* value. Observe that we cannot capture this information by simply adding another field to the Students schema. Thus, the ability to specify uniqueness of the values in a field increases the accuracy with which we can describe our data. The expressiveness of the constructs available for specifying integrity constraints is an important aspect of a data model.

Other Data Models

In addition to the relational data model (which is used in numerous systems, including IBM's DB2, Informix, Oracle, Sybase, Microsoft's Access, FoxBase, Paradox, Tandem, and Teradata), other important data models include the hierarchical model (e.g., used in IBM's IMS DBMS), the network model (e.g., used in IDS and IDMS), the object-oriented model (e.g., used in Objectstore and Versant), and the object-relational model (e.g., used in DBMS products from IBM, Informix, ObjectStore, Oracle, Versant, and others). While there are many databases that use the hierarchical and network models, and systems based on the object-oriented and object-relational models are gaining acceptance in the marketplace, the dominant model today is the relational model.

In this book, we will focus on the relational model because of its wide use and importance. Indeed, the object-relational model, which is gaining in popularity, is an effort to combine the best features of the relational and object-oriented models, and a good grasp of the relational model is necessary to understand object-relational concepts. (We discuss the object-oriented and object-relational models in Chapter 25.)

1.5.2 Levels of Abstraction in a DBMS

The data in a DBMS is described at three levels of abstraction, as illustrated in Figure 1.2. The database description consists of a schema at each of these three levels of abstraction: the *conceptual*, *physical*, and *external* schemas.

A **data definition language** (DDL) is used to define the external and conceptual schemas. We will discuss the DDL facilities of the most widely used database language, SQL, in Chapter 3. All DBMS vendors also support SQL commands to describe aspects of the physical schema, but these commands are not part of the SQL-92 language

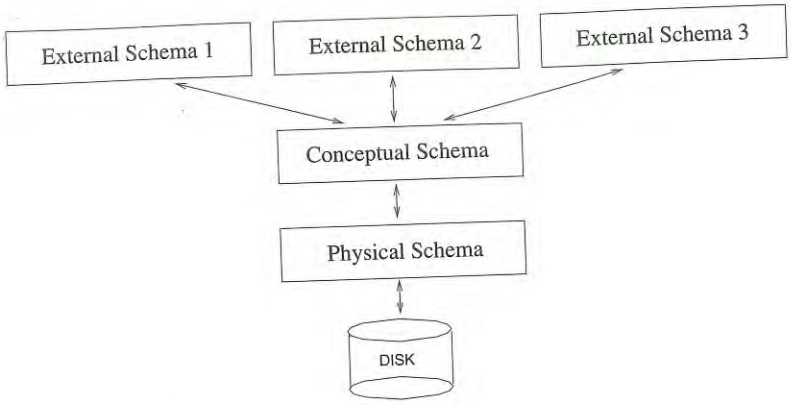


Figure 1.2 Levels of Abstraction in a DBMS

standard. Information about the conceptual, external, and physical schemas is stored in the **system catalogs** (Section 13.2). We discuss the three levels of abstraction in the rest of this section.

Conceptual Schema

The **conceptual schema** (sometimes called the **logical schema**) describes the stored data in terms of the data model of the DBMS. In a relational DBMS, the conceptual schema describes all relations that are stored in the database. In our sample university database, these relations contain information about *entities*, such as students and faculty, and about *relationships*, such as students' enrollment in courses. All student entities can be described using records in a Students relation, as we saw earlier. In fact, each collection of entities and each collection of relationships can be described as a relation, leading to the following conceptual schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`,
age: `integer`, *gpa:* `real`)
Faculty(*fid:* `string`, *fname:* `string`, *sal:* `real`)
Courses(*cid:* `string`, *cname:* `string`, *credits:* `integer`)
Rooms(*rno:* `integer`, *address:* `string`, *capacity:* `integer`)
Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`)
Teaches(*fid:* `string`, *cid:* `string`)
Meets_In(*cid:* `string`, *rno:* `integer`, *time:* `string`)

The choice of relations, and the choice of fields for each relation, is not always obvious, and the process of arriving at a good conceptual schema is called **conceptual database design**. We discuss conceptual database design in Chapters 2 and 15.

Physical Schema

The **physical schema** specifies additional storage details. Essentially, the physical schema summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices such as disks and tapes.

We must decide what file organizations to use to store the relations, and create auxiliary data structures called **indexes** to speed up data retrieval operations. A sample physical schema for the university database follows:

- Store all relations as unsorted files of records. (A file in a DBMS is either a collection of records or a collection of pages, rather than a string of characters as in an operating system.)

  Create indexes on the first column of the Students, Faculty, and Courses relations, the *sal* column of Faculty, and the *capacity* column of Rooms.

Decisions about the physical schema are based on an understanding of how the data is typically accessed. The process of arriving at a good physical schema is called **physical database design**. We discuss physical database design in Chapter 16.

External Schema

External schemas, which usually are also in terms of the data model of the DBMS, allow data access to be customized (and authorized) at the level of individual users or groups of users. Any given database has exactly one conceptual schema and one physical schema because it has just one set of stored relations, but it may have several external schemas, each tailored to a particular group of users. Each external schema consists of a collection of one or more **views** and relations from the conceptual schema. A view is conceptually a relation, but the records in a view are not stored in the DBMS. Rather, they are computed using a definition for the view, in terms of relations stored in the DBMS. We discuss views in more detail in Chapter 3.

The external schema design is guided by end user requirements. For example, we might want to allow students to find out the names of faculty members teaching courses, as well as course enrollments. This can be done by defining the following view:

Courseinfo(*cid:* `string`, *fname:* `string`, *enrollment:* `integer`)

A user can treat a view just like a relation and ask questions about the records in the view. Even though the records in the view are not stored explicitly, they are computed as needed. We did not include Courseinfo in the conceptual schema because we can compute Courseinfo from the relations in the conceptual schema, and to store it in addition would be redundant. Such redundancy, in addition to the wasted space, could

lead to inconsistencies. For example, a tuple may be inserted into the Enrolled relation, indicating that a particular student has enrolled in some course, without incrementing the value in the *enrollment* field of the corresponding record of Courseinfo (if the latter also is part of the conceptual schema and its tuples are stored in the DBMS).

1.5.3 Data Independence

A very important advantage of using a DBMS is that it offers **data independence**. That is, application programs are insulated from changes in the way the data is structured and stored. Data independence is achieved through use of the three levels of data abstraction; in particular, the conceptual schema and the external schema provide distinct benefits in this area.

Relations in the external schema (view relations) are in principle generated on demand from the relations corresponding to the conceptual schema.[3] If the underlying data is reorganized, that is, the conceptual schema is changed, the definition of a view relation can be modified so that the same relation is computed as before. For example, suppose that the Faculty relation in our university database is replaced by the following two relations:

Faculty_public(*fid:* `string`, *fname:* `string`, *office:* `integer`)
Faculty_private(*fid:* `string`, *sal:* `real`)

Intuitively, some confidential information about faculty has been placed in a separate relation and information about offices has been added. The Courseinfo view relation can be redefined in terms of Faculty_public and Faculty_private, which together contain all the information in Faculty, so that a user who queries Courseinfo will get the same answers as before.

Thus users can be shielded from changes in the logical structure of the data, or changes in the choice of relations to be stored. This property is called **logical data independence**.

In turn, the conceptual schema insulates users from changes in the physical storage of the data. This property is referred to as **physical data independence**. The conceptual schema hides details such as how the data is actually laid out on disk, the file structure, and the choice of indexes. As long as the conceptual schema remains the same, we can change these storage details without altering applications. (Of course, performance might be affected by such changes.)

[3] In practice, they could be precomputed and stored to speed up queries on view relations, but the computed view relations must be updated whenever the underlying relations are updated.

1.6 QUERIES IN A DBMS

The ease with which information can be obtained from a database often determines its value to a user. In contrast to older database systems, relational database systems allow a rich class of questions to be posed easily; this feature has contributed greatly to their popularity. Consider the sample university database in Section 1.5.2. Here are examples of questions that a user might ask:

1. What is the name of the student with student id 123456?
2. What is the average salary of professors who teach the course with cid CS564?
3. How many students are enrolled in course CS564?
4. What fraction of students in course CS564 received a grade better than B?
5. Is any student with a GPA less than 3.0 enrolled in course CS564?

Such questions involving the data stored in a DBMS are called **queries**. A DBMS provides a specialized language, called the **query language**, in which queries can be posed. A very attractive feature of the relational model is that it supports powerful query languages. **Relational calculus** is a formal query language based on mathematical logic, and queries in this language have an intuitive, precise meaning. **Relational algebra** is another formal query language, based on a collection of **operators** for manipulating relations, which is equivalent in power to the calculus.

A DBMS takes great care to evaluate queries as efficiently as possible. We discuss query optimization and evaluation in Chapters 12 and 13. Of course, the efficiency of query evaluation is determined to a large extent by how the data is stored physically. Indexes can be used to speed up many queries—in fact, a good choice of indexes for the underlying relations can speed up each query in the preceding list. We discuss data storage and indexing in Chapters 7, 8, 9, and 10.

A DBMS enables users to create, modify, and query data through a **data manipulation language** (DML). Thus, the query language is only one part of the DML, which also provides constructs to insert, delete, and modify data. We will discuss the DML features of SQL in Chapter 5. The DML and DDL are collectively referred to as the **data sublanguage** when embedded within a **host language** (e.g., C or COBOL).

1.7 TRANSACTION MANAGEMENT

Consider a database that holds information about airline reservations. At any given instant, it is possible (and likely) that several travel agents are looking up information about available seats on various flights and making new seat reservations. When several users access (and possibly modify) a database concurrently, the DBMS must order

their requests carefully to avoid conflicts. For example, when one travel agent looks up Flight 100 on some given day and finds an empty seat, another travel agent may simultaneously be making a reservation for that seat, thereby making the information seen by the first agent obsolete.

Another example of concurrent use is a bank's database. While one user's application program is computing the total deposits, another application may transfer money from an account that the first application has just 'seen' to an account that has not yet been seen, thereby causing the total to appear larger than it should be. Clearly, such anomalies should not be allowed to occur. However, disallowing concurrent access can degrade performance.

Further, the DBMS must protect users from the effects of system failures by ensuring that all data (and the status of active applications) is restored to a consistent state when the system is restarted after a crash. For example, if a travel agent asks for a reservation to be made, and the DBMS responds saying that the reservation has been made, the reservation should not be lost if the system crashes. On the other hand, if the DBMS has not yet responded to the request, but is in the process of making the necessary changes to the data while the crash occurs, the partial changes should be undone when the system comes back up.

A **transaction** is *any one execution* of a user program in a DBMS. (Executing the same program several times will generate several transactions.) This is the basic unit of change as seen by the DBMS: Partial transactions are not allowed, and the effect of a group of transactions is equivalent to some serial execution of all transactions. We briefly outline how these properties are guaranteed, deferring a detailed discussion to later chapters.

1.7.1 Concurrent Execution of Transactions

An important task of a DBMS is to schedule concurrent accesses to data so that each user can safely ignore the fact that others are accessing the data concurrently. The importance of this task cannot be underestimated because a database is typically shared by a large number of users, who submit their requests to the DBMS independently, and simply cannot be expected to deal with arbitrary changes being made concurrently by other users. A DBMS allows users to think of their programs as if they were executing in isolation, one after the other in some order chosen by the DBMS. For example, if a program that deposits cash into an account is submitted to the DBMS at the same time as another program that debits money from the same account, either of these programs could be run first by the DBMS, but their steps will not be interleaved in such a way that they interfere with each other.

A **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. A **lock** is a mechanism used to control access to database objects. Two kinds of **locks** are commonly supported by a DBMS: **shared locks** on an object can be held by two different transactions at the same time, but an **exclusive lock** on an object ensures that no other transactions hold *any* lock on this object.

Suppose that the following locking protocol is followed: Every transaction begins by obtaining a shared lock on each data object that it needs to read and an exclusive lock on each data object that it needs to modify, and then releases all its locks after completing all actions. Consider two transactions $T1$ and $T2$ such that $T1$ wants to modify a data object and $T2$ wants to read the same object. Intuitively, if $T1$'s request for an exclusive lock on the object is granted first, $T2$ cannot proceed until $T1$ releases this lock, because $T2$'s request for a shared lock will not be granted by the DBMS until then. Thus, all of $T1$'s actions will be completed before any of $T2$'s actions are initiated. We consider locking in more detail in Chapters 18 and 19.

1.7.2 Incomplete Transactions and System Crashes

Transactions can be interrupted before running to completion for a variety of reasons, e.g., a system crash. A DBMS must ensure that the changes made by such incomplete transactions are removed from the database. For example, if the DBMS is in the middle of transferring money from account A to account B, and has debited the first account but not yet credited the second when the crash occurs, the money debited from account A must be restored when the system comes back up after the crash.

To do so, the DBMS maintains a **log** of all writes to the database. A crucial property of the log is that each write action must be recorded in the log (on disk) *before* the corresponding change is reflected in the database itself—otherwise, if the system crashes just after making the change in the database but before the change is recorded in the log, the DBMS would be unable to detect and undo this change. This property is called **Write-Ahead Log** or **WAL**. To ensure this property, the DBMS must be able to selectively force a page in memory to disk.

The log is also used to ensure that the changes made by a successfully completed transaction are not lost due to a system crash, as explained in Chapter 20. Bringing the database to a consistent state after a system crash can be a slow process, since the DBMS must ensure that the effects of all transactions that completed prior to the crash are restored, and that the effects of incomplete transactions are undone. The time required to recover from a crash can be reduced by periodically forcing some information to disk; this periodic operation is called a **checkpoint**.

1.7.3 Points to Note

In summary, there are three points to remember with respect to DBMS support for concurrency control and recovery:

1. Every object that is read or written by a transaction is first locked in shared or exclusive mode, respectively. Placing a lock on an object restricts its availability to other transactions and thereby affects performance.
2. For efficient log maintenance, the DBMS must be able to selectively force a collection of pages in main memory to disk. Operating system support for this operation is not always satisfactory.
3. Periodic checkpointing can reduce the time needed to recover from a crash. Of course, this must be balanced against the fact that checkpointing too often slows down normal execution.

1.8 STRUCTURE OF A DBMS

Figure 1.3 shows the structure (with some simplification) of a typical DBMS based on the relational data model.

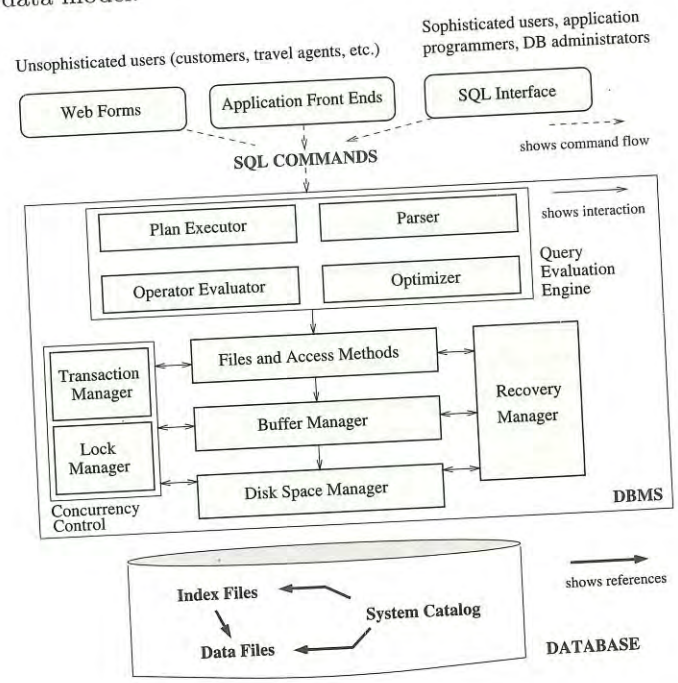


Figure 1.3 Architecture of a DBMS

The DBMS accepts SQL commands generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. (This is a simplification: SQL commands can be embedded in host-language application programs, e.g., Java or COBOL programs. We ignore these issues to concentrate on the core DBMS functionality.)

When a user issues a query, the parsed query is presented to a **query optimizer**, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An **execution plan** is a blueprint for evaluating a query, and is usually represented as a tree of relational operators (with annotations that contain additional detailed information about which access methods to use, etc.). We discuss query optimization in Chapter 13. Relational operators serve as the building blocks for evaluating queries posed against the data. The implementation of these operators is discussed in Chapter 12.

The code that implements relational operators sits on top of the file and access methods layer. This layer includes a variety of software for supporting the concept of a **file**, which, in a DBMS, is a collection of pages or a collection of records. This layer typically supports a **heap file**, or file of unordered pages, as well as indexes. In addition to keeping track of the pages in a file, this layer organizes the information within a page. File and page level storage issues are considered in Chapter 7. File organizations and indexes are considered in Chapter 8.

The files and access methods layer code sits on top of the **buffer manager**, which brings pages in from disk to main memory as needed in response to read requests. Buffer management is discussed in Chapter 7.

The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write pages through (routines provided by) this layer, called the **disk space manager**. This layer is discussed in Chapter 7.

The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. DBMS components associated with concurrency control and recovery include the **transaction manager**, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the **lock manager**, which keeps track of requests for locks and grants locks on database objects when they become available; and the **recovery manager**, which is responsible for maintaining a log, and restoring the system to a consistent state after a crash. The disk space manager, buffer manager, and file and access method layers must interact with these components. We discuss concurrency control and recovery in detail in Chapter 18.

1.9 PEOPLE WHO DEAL WITH DATABASES

Quite a variety of people are associated with the creation and use of databases. Obviously, there are **database implementors**, who build DBMS software, and **end users** who wish to store and use data in a DBMS. Database implementors work for vendors such as IBM or Oracle. End users come from a diverse and increasing number of fields. As data grows in complexity and volume, and is increasingly recognized as a major asset, the importance of maintaining it professionally in a DBMS is being widely accepted. Many end users simply use applications written by database application programmers (see below), and so require little technical knowledge about DBMS software. Of course, sophisticated users who make more extensive use of a DBMS, such as writing their own queries, require a deeper understanding of its features.

In addition to end users and implementors, two other classes of people are associated with a DBMS: *application programmers* and *database administrators* (DBAs).

Database application programmers develop packages that facilitate data access for end users, who are usually not computer professionals, using the host or data languages and software tools that DBMS vendors provide. (Such tools include report writers, spreadsheets, statistical packages, etc.) Application programs should ideally access data through the external schema. It is possible to write applications that access data at a lower level, but such applications would compromise data independence.

A personal database is typically maintained by the individual who owns it and uses it. However, corporate or enterprise-wide databases are typically important enough and complex enough that the task of designing and maintaining the database is entrusted to a professional called the **database administrator**. The DBA is responsible for many critical tasks:

- **Design of the conceptual and physical schemas:** The DBA is responsible for interacting with the users of the system to understand what data is to be stored in the DBMS and how it is likely to be used. Based on this knowledge, the DBA must design the conceptual schema (decide what relations to store) and the physical schema (decide how to store them). The DBA may also design widely used portions of the external schema, although users will probably augment this schema by creating additional views.

- **Security and authorization:** The DBA is responsible for ensuring that unauthorized data access is not permitted. In general, not everyone should be able to access all the data. In a relational DBMS, users can be granted permission to access only certain views and relations. For example, although you might allow students to find out course enrollments and who teaches a given course, you would not want students to see faculty salaries or each others' grade information.

The DBA can enforce this policy by giving students permission to read only the Courseinfo view.

- **Data availability and recovery from failures:** The DBA must take steps to ensure that if the system fails, users can continue to access as much of the uncorrupted data as possible. The DBA must also work to restore the data to a consistent state. The DBMS provides software support for these functions, but the DBA is responsible for implementing procedures to back up the data periodically and to maintain logs of system activity (to facilitate recovery from a crash).
- **Database tuning:** The needs of users are likely to evolve with time. The DBA is responsible for modifying the database, in particular the conceptual and physical schemas, to ensure adequate performance as user requirements change.

1.10 POINTS TO REVIEW

- A database management system (DBMS) is software that supports management of large collections of data. A DBMS provides efficient data access, data independence, data integrity, security, quick application development, support for concurrent access, and recovery from system failures. **(Section 1.1)**
- Storing data in a DBMS versus storing it in operating system files has many advantages. **(Section 1.3)**
- Using a DBMS provides the user with data independence, efficient data access, automatic data integrity, and security. **(Section 1.4)**
- The structure of the data is described in terms of a *data model* and the description is called a *schema*. The *relational model* is currently the most popular data model. A DBMS distinguishes between *external*, *conceptual*, and *physical schema* and thus allows a view of the data at three levels of abstraction. Physical and logical *data independence*, which are made possible by these three levels of abstraction, insulate the users of a DBMS from the way the data is structured and stored inside a DBMS. **(Section 1.5)**
- A *query language* and a *data manipulation language* enable high-level access and modification of the data. **(Section 1.6)**
- A *transaction* is a logical unit of access to a DBMS. The DBMS ensures that either all or none of a transaction's changes are applied to the database. For performance reasons, the DBMS processes multiple transactions concurently, but ensures that the result is equivalent to running the transactions one after the other in some order. The DBMS maintains a record of all changes to the data in the *system log*, in order to undo partial transactions and recover from system crashes. *Checkpointing* is a periodic operation that can reduce the time for recovery from a crash. **(Section 1.7)**

- DBMS code is organized into several modules: the disk space manager, the buffer manager, a layer that supports the abstractions of files and index structures, a layer that implements relational operators, and a layer that optimizes queries and produces an execution plan in terms of relational operators. **(Section 1.8)**
- A *database administrator (DBA)* manages a DBMS for an enterprise. The DBA designs schemas, provide security, restores the system after a failure, and periodically tunes the database to meet changing user needs. Application programmers develop applications that use DBMS functionality to access and manipulate data, and end users invoke these applications. **(Section 1.9)**

EXERCISES

Exercise 1.1 Why would you choose a database system instead of simply storing data in operating system files? When would it make sense *not* to use a database system?

Exercise 1.2 What is logical data independence and why is it important?

Exercise 1.3 Explain the difference between logical and physical data independence.

Exercise 1.4 Explain the difference between external, internal, and conceptual schemas. How are these different schema layers related to the concepts of logical and physical data independence?

Exercise 1.5 What are the responsibilities of a DBA? If we assume that the DBA is never interested in running his or her own queries, does the DBA still need to understand query optimization? Why?

Exercise 1.6 Scrooge McNugget wants to store information (names, addresses, descriptions of embarrassing moments, etc.) about the many ducks on his payroll. Not surprisingly, the volume of data compels him to buy a database system. To save money, he wants to buy one with the fewest possible features, and he plans to run it as a stand-alone application on his PC clone. Of course, Scrooge does not plan to share his list with anyone. Indicate which of the following DBMS features Scrooge should pay for; in each case also indicate why Scrooge should (or should not) pay for that feature in the system he buys.

1. A security facility.
2. Concurrency control.
3. Crash recovery.
4. A view mechanism.
5. A query language.

Exercise 1.7 Which of the following plays an important role in *representing* information about the real world in a database? Explain briefly.

1. The data definition language.

2. The data manipulation language.
3. The buffer manager.
4. The data model.

Exercise 1.8 Describe the structure of a DBMS. If your operating system is upgraded to support some new functions on OS files (e.g., the ability to force some sequence of bytes to disk), which layer(s) of the DBMS would you have to rewrite in order to take advantage of these new functions?

Exercise 1.9 Answer the following questions:

1. What is a transaction?
2. Why does a DBMS interleave the actions of different transactions, instead of executing transactions one after the other?
3. What must a user guarantee with respect to a transaction and database consistency? What should a DBMS guarantee with respect to concurrent execution of several transactions and database consistency?
4. Explain the strict two-phase locking protocol.
5. What is the WAL property, and why is it important?

PROJECT-BASED EXERCISES

Exercise 1.10 Use a Web browser to look at the HTML documentation for Minibase. Try to get a feel for the overall architecture.

BIBLIOGRAPHIC NOTES

The evolution of database management systems is traced in [248]. The use of data models for describing real-world data is discussed in [361], and [363] contains a taxonomy of data models. The three levels of abstraction were introduced in [155, 623]. The network data model is described in [155], and [680] discusses several commercial systems based on this model. [634] contains a good annotated collection of systems-oriented papers on database management.

Other texts covering database management systems include [169, 208, 289, 600, 499, 656, 669]. [169] provides a detailed discussion of the relational model from a conceptual standpoint and is notable for its extensive annotated bibliography. [499] presents a performance-oriented perspective, with references to several commercial systems. [208] and [600] offer broad coverage of database system concepts, including a discussion of the hierarchical and network data models. [289] emphasizes the connection between database query languages and logic programming. [669] emphasizes data models. Of these texts, [656] provides the most detailed discussion of theoretical issues. Texts devoted to theoretical aspects include [38, 436, 3]. Handbook [653] includes a section on databases that contains introductory survey articles on a number of topics.

2

THE ENTITY-RELATIONSHIP MODEL

> The great successful men of the world have used their imaginations. They think ahead and create their mental picture, and then go to work materializing that picture in all its details, filling in here, adding a little there, altering this bit and that bit, but steadily building, steadily building.
>
> —Robert Collier

The *entity-relationship (ER) data model* allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design. In this chapter, we introduce the ER model and discuss how its features allow us to model a wide range of data faithfully.

The ER model is important primarily for its role in database design. It provides useful concepts that allow us to move from an informal description of what users want from their database to a more detailed, and precise, description that can be implemented in a DBMS. We begin with an overview of database design in Section 2.1 in order to motivate our discussion of the ER model. Within the larger context of the overall design process, the ER model is used in a phase called *conceptual database design*. We then introduce the ER model in Sections 2.2, 2.3, and 2.4. In Section 2.5, we discuss database design issues involving the ER model. We conclude with a brief discussion of conceptual database design for large enterprises.

We note that many variations of ER diagrams are in use, and no widely accepted standards prevail. The presentation in this chapter is representative of the family of ER models and includes a selection of the most popular features.

2.1 OVERVIEW OF DATABASE DESIGN

The database design process can be divided into six steps. The ER model is most relevant to the first three steps:

(1) Requirements Analysis: The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database.

Database design tools: Design tools are available from RDBMS vendors as well as third-party vendors. Sybase and Oracle, in particular, have comprehensive sets design and analysis tools. See the following URL for details on Sybase's tools: `http://www.sybase.com/products/application_tools` The following provides details on Oracle's tools: `http://www.oracle.com/tools`

This is usually an informal process that involves discussions with user groups, a study of the current operating environment and how it is expected to change, analysis of any available documentation on existing applications that are expected to be replaced or complemented by the database, and so on. Several methodologies have been proposed for organizing and presenting the information gathered in this step, and some automated tools have been developed to support this process.

(2) Conceptual Database Design: The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model, and is discussed in the rest of this chapter.

(3) Logical Database Design: We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will only consider relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema. We discuss this step in detail in Chapter 3; the result is a conceptual schema, sometimes called the **logical schema**, in the relational data model.

2.1.1 Beyond the ER Model

ER modeling is sometimes regarded as a complete approach to designing a logical database schema. This is incorrect because the ER diagram is just an approximate description of the data, constructed through a very subjective evaluation of the information collected during requirements analysis. A more careful analysis can often refine the logical schema obtained at the end of Step 3. Once we have a good logical schema, we must consider performance criteria and design the physical schema. Finally, we must address security issues and ensure that users are able to access the data they need, but not data that we wish to hide from them. The remaining three steps of database design are briefly described below: [1]

[1]This material can be omitted on a first reading of this chapter without loss of continuity.

(4) Schema Refinement: The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory. We discuss the theory of *normalizing* relations—restructuring them to ensure some desirable properties—in Chapter 15.

(5) Physical Database Design: In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps. We discuss physical design and database tuning in Chapter 16.

(6) Security Design: In this step, we identify different user groups and different **roles** played by various users (e.g., the development team for a product, the customer support representatives, the product manager). For each role and user group, we must identify the parts of the database that they must be able to access and the parts of the database that they should *not* be allowed to access, and take steps to ensure that they can access only the necessary parts. A DBMS provides several mechanisms to assist in this step, and we discuss this in Chapter 17.

In general, our division of the design process into steps should be seen as a classification of the *kinds* of steps involved in design. Realistically, although we might begin with the six step process outlined here, a complete database design will probably require a subsequent **tuning** phase in which all six kinds of design steps are interleaved and repeated until the design is satisfactory. Further, we have omitted the important steps of implementing the database design, and designing and implementing the application layers that run on top of the DBMS. In practice, of course, these additional steps can lead to a rethinking of the basic database design.

The concepts and techniques that underlie a relational DBMS are clearly useful to someone who wants to implement or maintain the internals of a database system. However, it is important to recognize that serious users and DBAs must also know how a DBMS works. A good understanding of database system internals is essential for a user who wishes to take full advantage of a DBMS and design a good database; this is especially true of physical design and database tuning.

2.2 ENTITIES, ATTRIBUTES, AND ENTITY SETS

An **entity** is an object in the real world that is distinguishable from other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy depart-

ment. It is often useful to identify a collection of similar entities. Such a collection is called an **entity set**. Note that entity sets need not be disjoint; the collection of toy department employees and the collection of appliance department employees may both contain employee John Doe (who happens to work in both departments). We could also define an entity set called Employees that contains both the toy and appliance department employee sets.

An entity is described using a set of **attributes**. All entities in a given entity set have the same attributes; this is essentially what we mean by *similar*. (This statement is an oversimplification, as we will see when we discuss inheritance hierarchies in Section 2.4.4, but it suffices for now and highlights the main idea.) Our choice of attributes reflects the level of detail at which we wish to represent information about entities. For example, the Employees entity set could use name, social security number (ssn), and parking lot (lot) as attributes. In this case we will store the name, social security number, and lot number for each employee. However, we will not store, say, an employee's address (or gender or age).

For each attribute associated with an entity set, we must identify a **domain** of possible values. For example, the domain associated with the attribute *name* of Employees might be the set of 20-character strings.[2] As another example, if the company rates employees on a scale of 1 to 10 and stores ratings in a field called *rating*, the associated domain consists of integers 1 through 10. Further, for each entity set, we choose a *key*. A **key** is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one **candidate** key; if so, we designate one of them as the **primary** key. For now we will assume that each entity set contains at least one set of attributes that uniquely identifies an entity in the entity set; that is, the set of attributes contains a key. We will revisit this point in Section 2.4.3.

The Employees entity set with attributes *ssn*, *name*, and *lot* is shown in Figure 2.1. An entity set is represented by a rectangle, and an attribute is represented by an oval. Each attribute in the primary key is underlined. The domain information could be listed along with the attribute name, but we omit this to keep the figures compact. The key is *ssn*.

2.3 RELATIONSHIPS AND RELATIONSHIP SETS

A **relationship** is an association among two or more entities. For example, we may have the relationship that Attishoo works in the pharmacy department. As with entities, we may wish to collect a set of similar relationships into a **relationship set**.

[2]To avoid confusion, we will assume that attribute names do not repeat across entity sets. This is not a real limitation because we can always use the entity set name to resolve ambiguities if the same attribute name is used in more than one entity set.

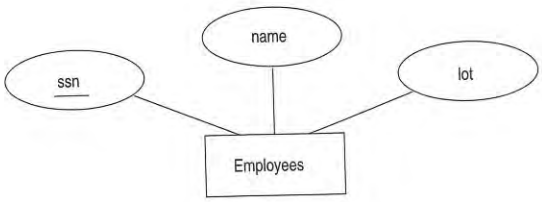


Figure 2.1 The Employees Entity Set

A relationship set can be thought of as a set of n-tuples:

$$\{(e_1, \ldots, e_n) \mid e_1 \in E_1, \ldots, e_n \in E_n\}$$

Each n-tuple denotes a relationship involving n entities e_1 through e_n, where entity e_i is in entity set E_i. In Figure 2.2 we show the relationship set Works_In, in which each relationship indicates a department in which an employee works. Note that several relationship sets might involve the same entity sets. For example, we could also have a Manages relationship set involving Employees and Departments.

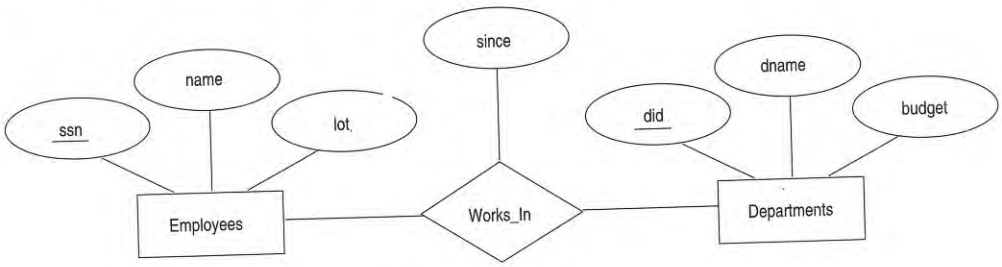


Figure 2.2 The Works_In Relationship Set

A relationship can also have **descriptive attributes**. Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities; for example, we may wish to record that Attishoo works in the pharmacy department as of January 1991. This information is captured in Figure 2.2 by adding an attribute, *since*, to Works_In. A relationship must be uniquely identified by the participating entities, without reference to the descriptive attributes. In the Works_In relationship set, for example, each Works_In relationship must be uniquely identified by the combination of employee *ssn* and department *did*. Thus, for a given employee-department pair, we cannot have more than one associated *since* value.

An **instance** of a relationship set is a set of relationships. Intuitively, an instance can be thought of as a 'snapshot' of the relationship set at some instant in time. An instance of the Works_In relationship set is shown in Figure 2.3. Each Employees entity is denoted by its *ssn*, and each Departments entity is denoted by its *did*, for simplicity.

The *since* value is shown beside each relationship. (The 'many-to-many' and 'total participation' comments in the figure will be discussed later, when we discuss integrity constraints.)

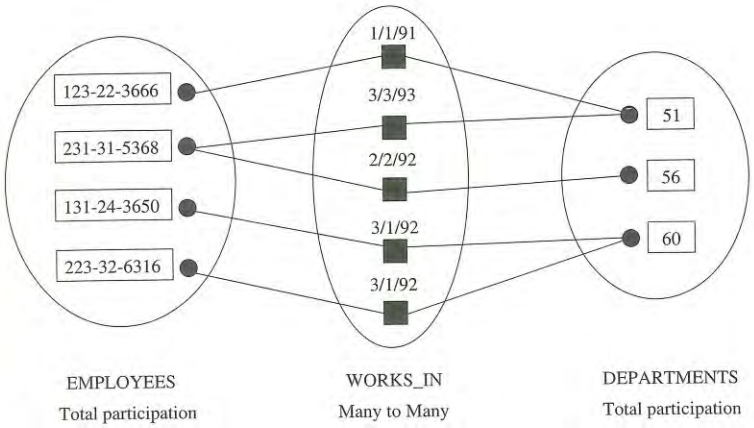


Figure 2.3 An Instance of the Works_In Relationship Set

As another example of an ER diagram, suppose that each department has offices in several locations and we want to record the locations at which each employee works. This relationship is **ternary** because we must record an association between an employee, a department, and a location. The ER diagram for this variant of Works_In, which we call Works_In2, is shown in Figure 2.4.

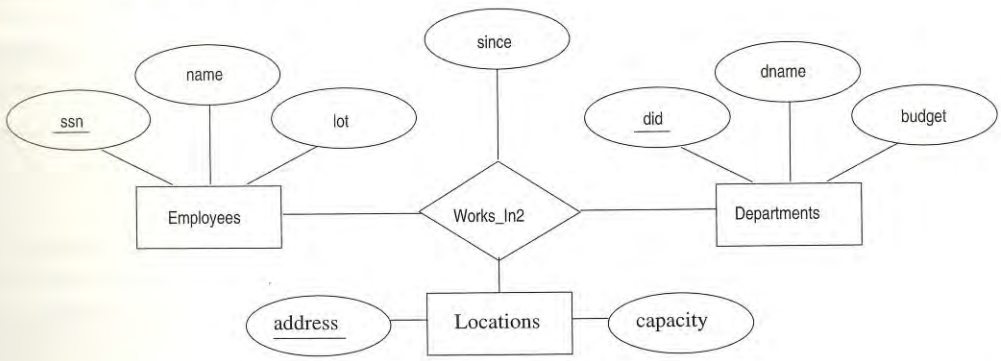


Figure 2.4 A Ternary Relationship Set

The entity sets that participate in a relationship set need not be distinct; sometimes a relationship might involve two entities in the same entity set. For example, consider the Reports_To relationship set that is shown in Figure 2.5. Since employees report to other employees, every relationship in Reports_To is of the form (emp_1, emp_2),

where both emp_1 and emp_2 are entities in Employees. However, they play different **roles**: emp_1 reports to the managing employee emp_2, which is reflected in the **role indicators** *supervisor* and *subordinate* in Figure 2.5. If an entity set plays more than one role, the role indicator concatenated with an attribute name from the entity set gives us a unique name for each attribute in the relationship set. For example, the Reports_To relationship set has attributes corresponding to the *ssn* of the supervisor and the *ssn* of the subordinate, and the names of these attributes are *supervisor_ssn* and *subordinate_ssn*.

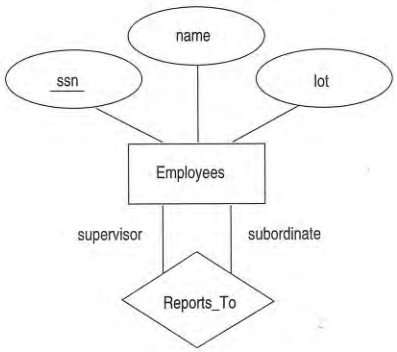


Figure 2.5 The Reports_To Relationship Set

2.4 ADDITIONAL FEATURES OF THE ER MODEL

We now look at some of the constructs in the ER model that allow us to describe some subtle properties of the data. The expressiveness of the ER model is a big reason for its widespread use.

2.4.1 Key Constraints

Consider the Works_In relationship shown in Figure 2.2. An employee can work in several departments, and a department can have several employees, as illustrated in the Works_In instance shown in Figure 2.3. Employee 231-31-5368 has worked in Department 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 has two employees.

Now consider another relationship set called Manages between the Employees and Departments entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department. The restriction that each department has at most one manager is an example of a **key constraint**, and it implies that each Departments entity appears in at most one Manages relationship

in any allowable instance of Manages. This restriction is indicated in the ER diagram of Figure 2.6 by using an arrow from Departments to Manages. Intuitively, the arrow states that given a Departments entity, we can uniquely determine the Manages relationship in which it appears.

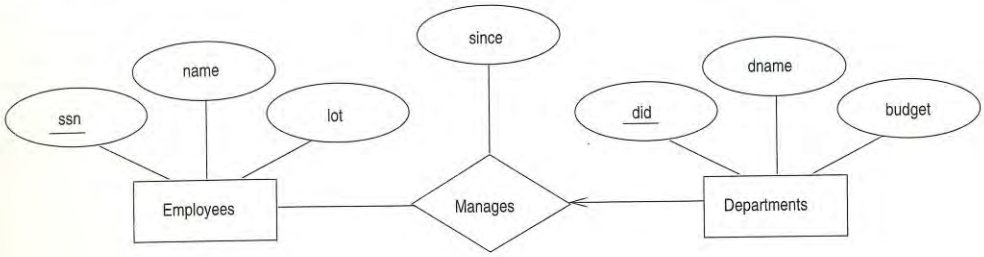


Figure 2.6 Key Constraint on Manages

An instance of the Manages relationship set is shown in Figure 2.7. While this is also a potential instance for the Works_In relationship set, the instance of Works_In shown in Figure 2.3 violates the key constraint on Manages.

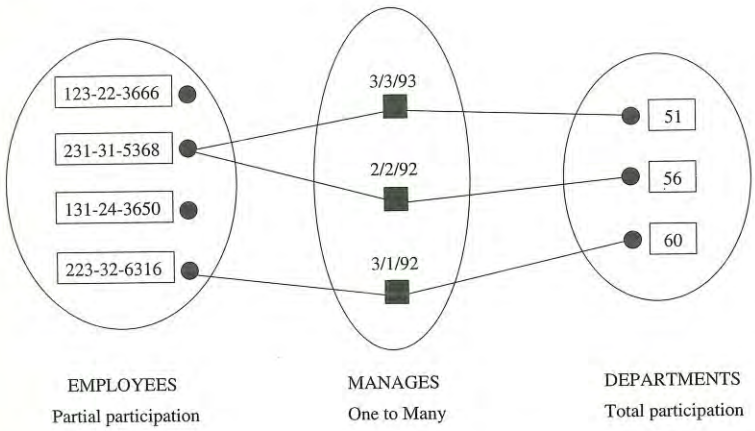


Figure 2.7 An Instance of the Manages Relationship Set

A relationship set like Manages is sometimes said to be **one-to-many**, to indicate that *one* employee can be associated with *many* departments (in the capacity of a manager), whereas each department can be associated with at most one employee as its manager. In contrast, the Works_In relationship set, in which an employee is allowed to work in several departments and a department is allowed to have several employees, is said to be **many-to-many**.

If we add the restriction that each employee can manage at most one department to the Manages relationship set, which would be indicated by adding an arrow from Employees to Manages in Figure 2.6, we have a **one-to-one** relationship set.

Key Constraints for Ternary Relationships

We can extend this convention—and the underlying key constraint concept—to relationship sets involving three or more entity sets: If an entity set E has a key constraint in a relationship set R, each entity in an instance of E appears in at most one relationship in (a corresponding instance of) R. To indicate a key constraint on entity set E in relationship set R, we draw an arrow from E to R.

In Figure 2.8, we show a ternary relationship with key constraints. Each employee works in at most one department, and at a single location. An instance of the Works_In3 relationship set is shown in Figure 2.9. Notice that each department can be associated with several employees and locations, and each location can be associated with several departments and employees; however, each employee is associated with a single department and location.

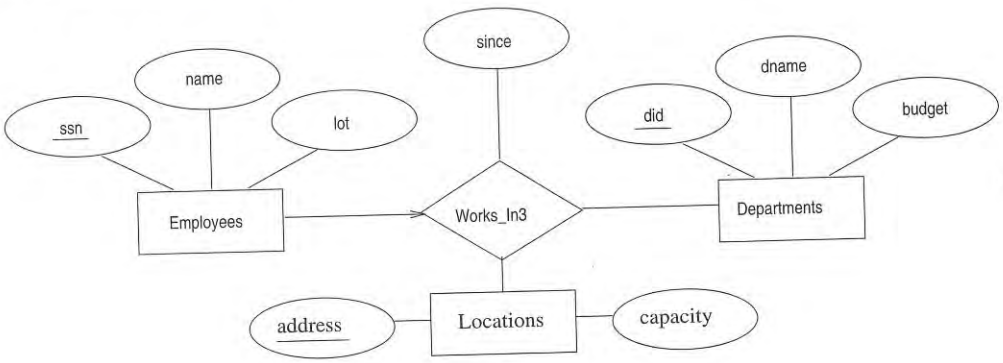


Figure 2.8 A Ternary Relationship Set with Key Constraints

2.4.2 Participation Constraints

The key constraint on Manages tells us that a department has at most one manager. A natural question to ask is whether every department has a manager. Let us say that every department is required to have a manager. This requirement is an example of a **participation constraint**; the participation of the entity set Departments in the relationship set Manages is said to be **total**. A participation that is not total is said to be **partial**. As an example, the participation of the entity set Employees in Manages is partial, since not every employee gets to manage a department.

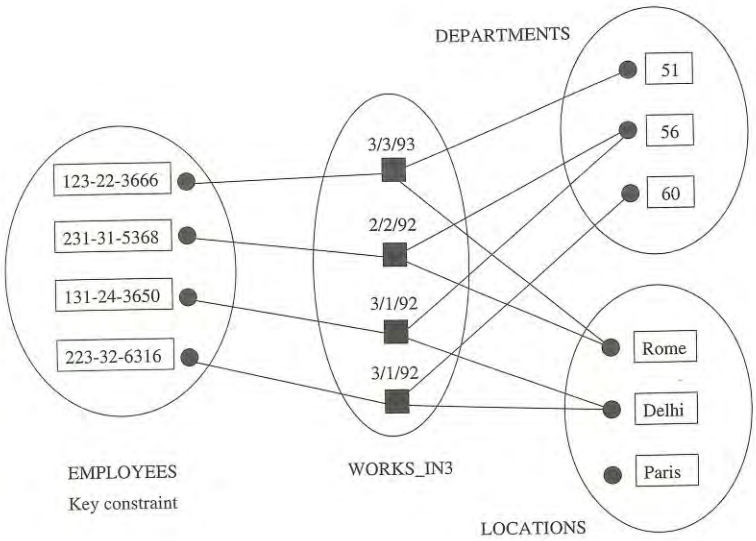


Figure 2.9 An Instance of Works_In3

Revisiting the Works_In relationship set, it is natural to expect that each employee works in at least one department and that each department has at least one employee. This means that the participation of both Employees and Departments in Works_In is total. The ER diagram in Figure 2.10 shows both the Manages and Works_In relationship sets and all the given constraints. If the participation of an entity set in a relationship set is total, the two are connected by a thick line; independently, the presence of an arrow indicates a key constraint. The instances of Works_In and Manages shown in Figures 2.3 and 2.7 satisfy all the constraints in Figure 2.10.

2.4.3 Weak Entities

Thus far, we have assumed that the attributes associated with an entity set include a key. This assumption does not always hold. For example, suppose that employees can purchase insurance policies to cover their dependents. We wish to record information about policies, including who is covered by each policy, but this information is really our only interest in the dependents of an employee. If an employee quits, any policy owned by the employee is terminated and we want to delete all the relevant policy and dependent information from the database.

We might choose to identify a dependent by name alone in this situation, since it is reasonable to expect that the dependents of a given employee have different names. Thus the attributes of the Dependents entity set might be *pname* and *age*. The attribute *pname* does *not* identify a dependent uniquely. Recall that the key for Employees is

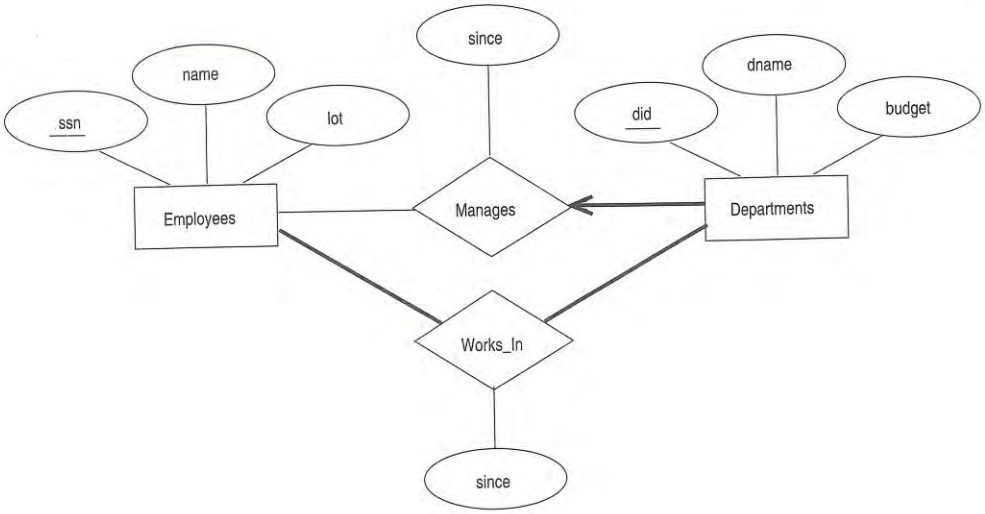


Figure 2.10 Manages and Works_In

ssn; thus we might have two employees called Smethurst, and each might have a son called Joe.

Dependents is an example of a **weak entity set**. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the **identifying owner**.

The following restrictions must hold:

- The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the **identifying relationship set** of the weak entity set.
- The weak entity set must have total participation in the identifying relationship set.

For example, a Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity. The set of attributes of a weak entity set that uniquely identify a weak entity for a given owner entity is called a *partial key* of the weak entity set. In our example *pname* is a partial key for Dependents.

The Dependents weak entity set and its relationship to Employees is shown in Figure 2.11. The total participation of Dependents in Policy is indicated by linking them

with a dark line. The arrow from Dependents to Policy indicates that each Dependents entity appears in at most one (indeed, exactly one, because of the participation constraint) Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with dark lines. To indicate that *pname* is a partial key for Dependents, we underline it using a broken line. This means that there may well be two dependents with the same *pname* value.

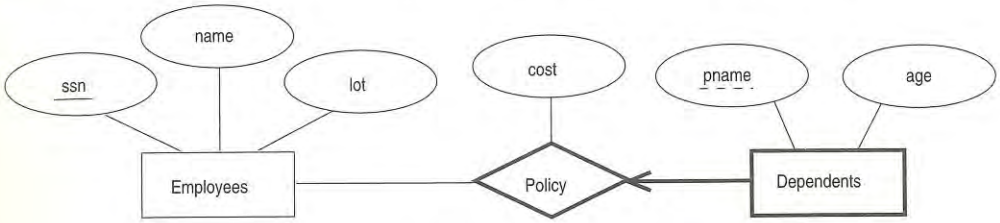


Figure 2.11 A Weak Entity Set

2.4.4 Class Hierarchies

Sometimes it is natural to classify the entities in an entity set into subclasses. For example, we might want to talk about an Hourly_Emps entity set and a Contract_Emps entity set to distinguish the basis on which they are paid. We might have attributes *hours_worked* and *hourly_wage* defined for Hourly_Emps and an attribute *contractid* defined for Contract_Emps.

We want the semantics that every entity in one of these sets is also an Employees entity, and as such must have all of the attributes of Employees defined. Thus, the attributes defined for an Hourly_Emps entity are the attributes for Employees plus Hourly_Emps. We say that the attributes for the entity set Employees are **inherited** by the entity set Hourly_Emps, and that Hourly_Emps **ISA** (read *is a*) Employees. In addition—and in contrast to class hierarchies in programming languages such as C++—there is a constraint on queries over instances of these entity sets: A query that asks for all Employees entities must consider all Hourly_Emps and Contract_Emps entities as well. Figure 2.12 illustrates the class hierarchy.

The entity set Employees may also be classified using a different criterion. For example, we might identify a subset of employees as Senior_Emps. We can modify Figure 2.12 to reflect this change by adding a second ISA node as a child of Employees and making Senior_Emps a child of this node. Each of these entity sets might be classified further, creating a multilevel ISA hierarchy.

A class hierarchy can be viewed in one of two ways:

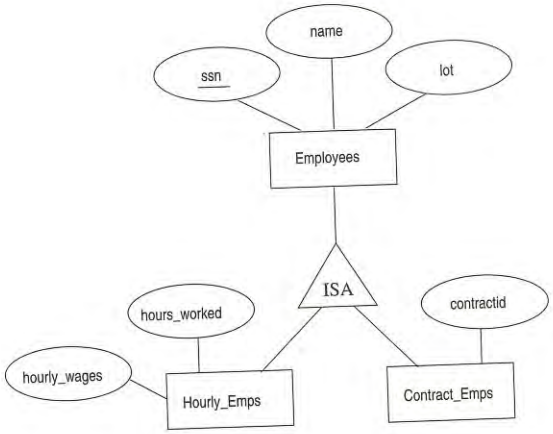


Figure 2.12 Class Hierarchy

- Employees is **specialized** into subclasses. Specialization is the process of identifying subsets of an entity set (the **superclass**) that share some distinguishing characteristic. Typically the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added.

- Hourly_Emps and Contract_Emps are **generalized** by Employees. As another example, two entity sets Motorboats and Cars may be generalized into an entity set Motor_Vehicles. Generalization consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.

We can specify two kinds of constraints with respect to ISA hierarchies, namely, *overlap* and *covering* constraints. **Overlap constraints** determine whether two subclasses are allowed to contain the same entity. For example, can Attishoo be both an Hourly_Emps entity and a Contract_Emps entity? Intuitively, no. Can he be both a Contract_Emps entity and a Senior_Emps entity? Intuitively, yes. We denote this by writing 'Contract_Emps `OVERLAPS` Senior_Emps.' In the absence of such a statement, we assume by default that entity sets are constrained to have no overlap.

Covering constraints determine whether the entities in the subclasses collectively include all entities in the superclass. For example, does every Employees entity have to belong to one of its subclasses? Intuitively, no. Does every Motor_Vehicles entity have to be either a Motorboats entity or a Cars entity? Intuitively, yes; a characteristic property of generalization hierarchies is that every instance of a superclass is an instance of a subclass. We denote this by writing 'Motorboats `AND` Cars `COVER`

Motor_Vehicles.' In the absence of such a statement, we assume by default that there is no covering constraint; we can have motor vehicles that are not motorboats or cars.

There are two basic reasons for identifying subclasses (by specialization or generalization):

1. We might want to add descriptive attributes that make sense only for the entities in a subclass. For example, *hourly_wages* does not make sense for a Contract_Emps entity, whose pay is determined by an individual contract.

2. We might want to identify the set of entities that participate in some relationship. For example, we might wish to define the Manages relationship so that the participating entity sets are Senior_Emps and Departments, to ensure that only senior employees can be managers. As another example, Motorboats and Cars may have different descriptive attributes (say, tonnage and number of doors), but as Motor_Vehicles entities, they must be licensed. The licensing information can be captured by a Licensed_To relationship between Motor_Vehicles and an entity set called Owners.

2.4.5 Aggregation

As we have defined it thus far, a relationship set is an association between entity sets. Sometimes we have to model a relationship between a collection of entities and *relationships*. Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more *entities*.

In order to define a relationship set such as Monitors, we introduce a new feature of the ER model, called *aggregation*. **Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set. This is illustrated in Figure 2.13, with a dashed box around Sponsors (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the Monitors relationship set.

When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can't we express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the

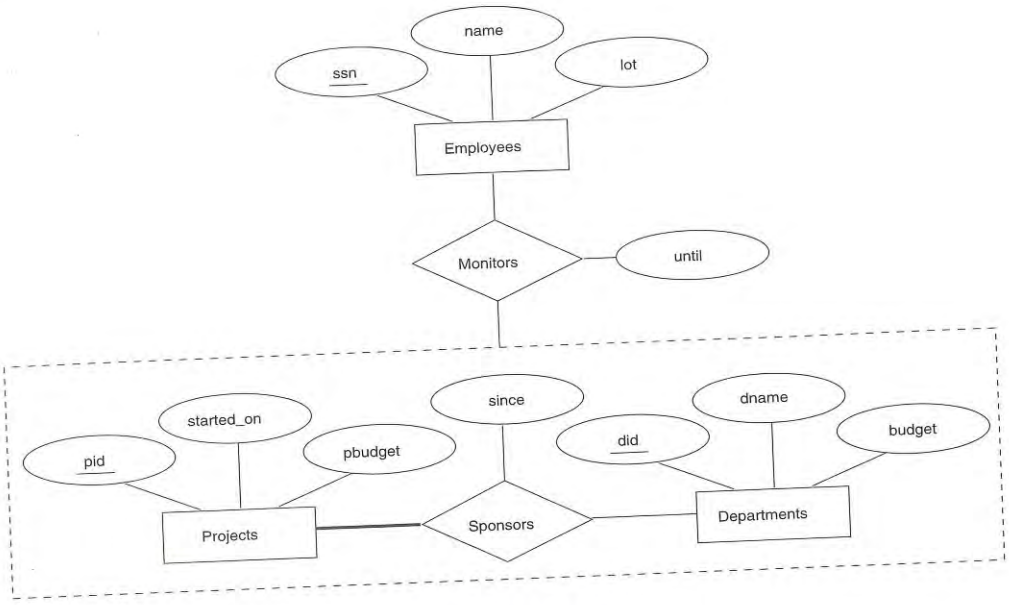


Figure 2.13 Aggregation

Monitors relationship has an attribute *until* that records the date until when the employee is appointed as the sponsorship monitor. Compare this attribute with the attribute *since* of Sponsors, which is the date when the sponsorship took effect. The use of aggregation versus a ternary relationship may also be guided by certain integrity constraints, as explained in Section 2.5.4.

2.5 CONCEPTUAL DATABASE DESIGN WITH THE ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
- Should we use aggregation?

We now discuss the issues involved in making these choices.

2.5.1 Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set (and related to the first entity set using a relationship set). For example, consider adding address information to the Employees entity set. One option is to use an attribute *address*. This option is appropriate if we need to record only one address per employee, and it suffices to think of an address as a string. An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship (say, Has_Address). This more complex alternative is necessary in two situations:

- We have to record more than one address for an employee.
- We want to capture the structure of an address in our ER diagram. For example, we might break down an address into city, state, country, and Zip code, in addition to a string for street information. By representing an address as an entity with these attributes, we can support queries such as "Find all employees with an address in Madison, WI."

For another example of when to model a concept as an entity set rather than as an attribute, consider the relationship set (called Works_In2) shown in Figure 2.14.

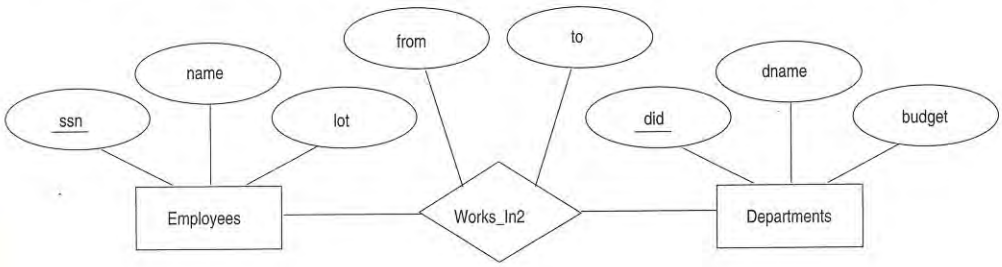


Figure 2.14 The Works_In2 Relationship Set

It differs from the Works_In relationship set of Figure 2.2 only in that it has attributes *from* and *to*, instead of *since*. Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram's semantics. The problem is that we want to record several values for the descriptive attributes for each instance of the Works_In2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes *from* and *to*, as shown in Figure 2.15.

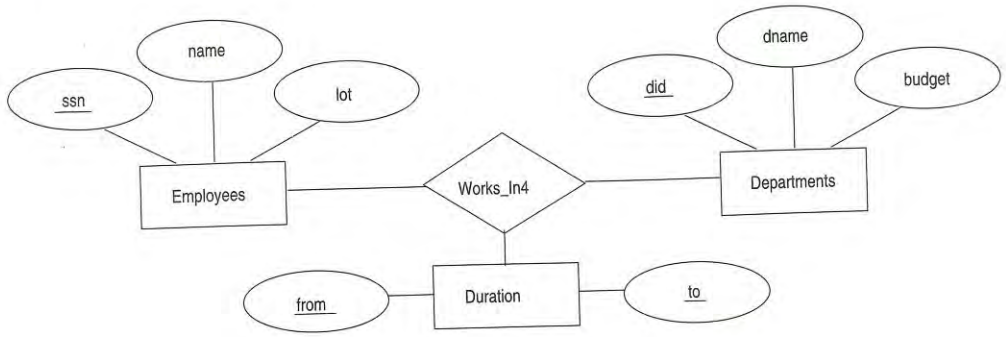


Figure 2.15 The Works_In4 Relationship Set

In some versions of the ER model, attributes are allowed to take on sets as values. Given this feature, we could make Duration an attribute of Works_In, rather than an entity set; associated with each Works_In relationship, we would have a set of intervals. This approach is perhaps more intuitive than modeling Duration as an entity set. Nonetheless, when such set-valued attributes are translated into the relational model, which does not support set-valued attributes, the resulting relational schema is very similar to what we get by regarding Duration as an entity set.

2.5.2 Entity versus Relationship

Consider the relationship set called Manages in Figure 2.6. Suppose that each department manager is given a discretionary budget (*dbudget*), as shown in Figure 2.16, in which we have also renamed the relationship set to Manages2.

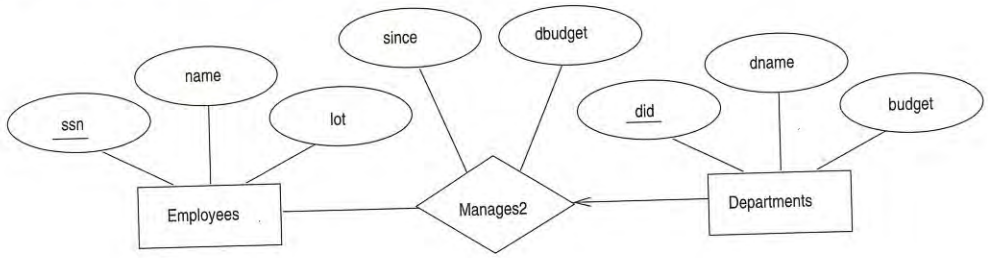


Figure 2.16 Entity versus Relationship

There is at most one employee managing a department, but a given employee could manage several departments; we store the starting date and discretionary budget for each manager-department pair. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.

But what if the discretionary budget is a sum that covers *all* departments managed by that employee? In this case each Manages2 relationship that involves a given employee will have the same value in the *dbudget* field. In general such redundancy could be significant and could cause a variety of problems. (We discuss redundancy and its attendant problems in Chapter 15.) Another problem with this design is that it is misleading.

We can address these problems by associating *dbudget* with the appointment of the employee as manager of a *group* of departments. In this approach, we model the appointment as an entity set, say Mgr_Appt, and use a ternary relationship, say Manages3, to relate a manager, an appointment, and a department. The details of an appointment (such as the discretionary budget) are not repeated for each department that is included in the appointment now, although there is still one Manages3 relationship instance per such department. Further, note that each department has at most one manager, as before, because of the key constraint. This approach is illustrated in Figure 2.17.

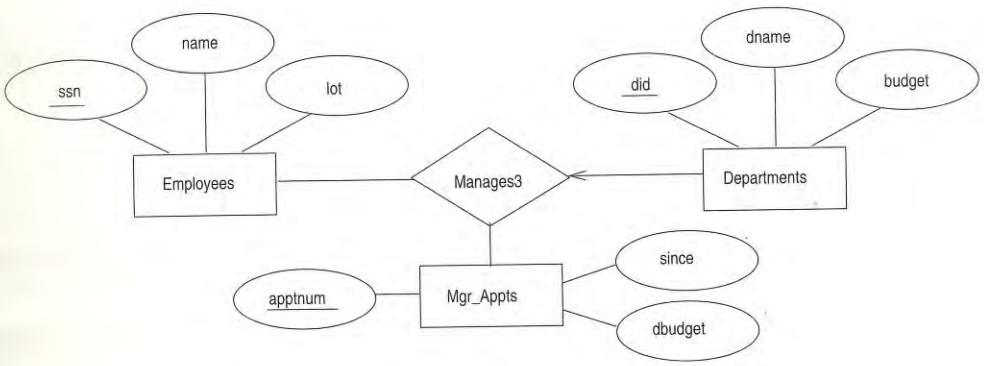


Figure 2.17 Entity Set versus Relationship

2.5.3 Binary versus Ternary Relationships *

Consider the ER diagram shown in Figure 2.18. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.

Suppose that we have the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.

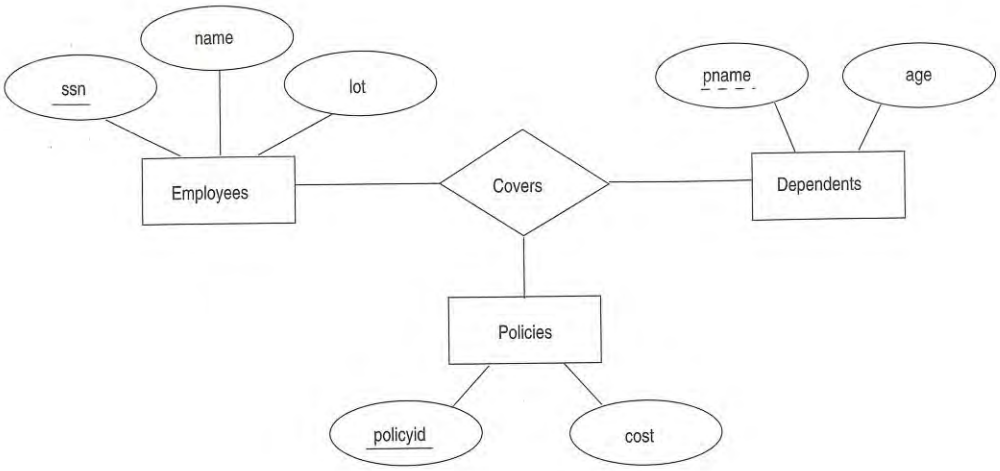


Figure 2.18 Policies as an Entity Set

- Dependents is a weak entity set, and each dependent entity is uniquely identified by taking *pname* in conjunction with the *policyid* of a policy entity (which, intuitively, covers the given dependent).

The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).

Even ignoring the third point above, the best way to model this situation is to use two binary relationships, as shown in Figure 2.19.

This example really had two relationships involving Policies, and our attempt to use a single ternary relationship (Figure 2.18) was inappropriate. There are situations, however, where a relationship inherently associates more than two entities. We have seen such an example in Figure 2.4 and also Figures 2.15 and 2.17.

As a good example of a ternary relationship, consider entity sets Parts, Suppliers, and Departments, and a relationship set Contracts (with descriptive attribute *qty*) that involves all of them. A contract specifies that a supplier will supply (some quantity of) a part to a department. This relationship cannot be adequately captured by a collection of binary relationships (without the use of aggregation). With binary relationships, we can denote that a supplier 'can supply' certain parts, that a department 'needs' some

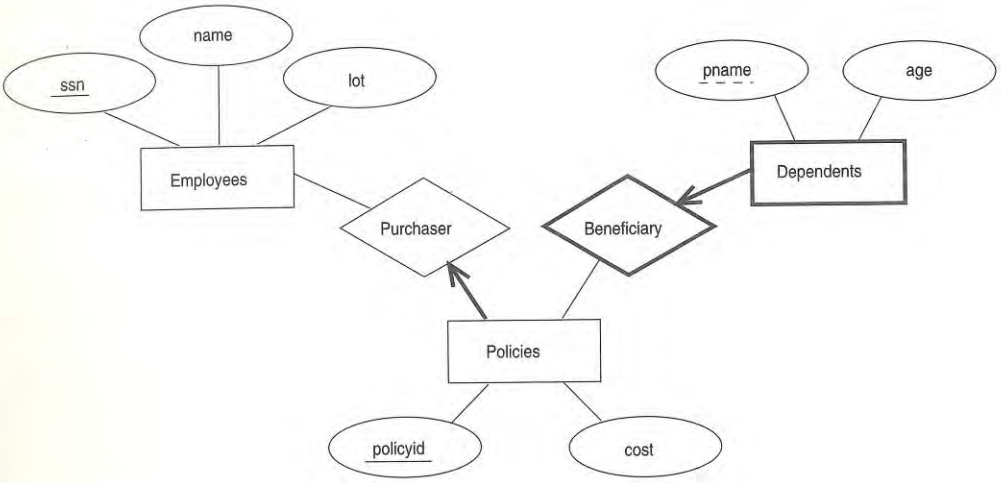


Figure 2.19 Policy Revisited

parts, or that a department 'deals with' a certain supplier. No combination of these relationships expresses the meaning of a contract adequately, for at least two reasons:

- The facts that supplier S can supply part P, that department D needs part P, and that D will buy from S do not necessarily imply that department D indeed buys part P from supplier S!
- We cannot represent the *qty* attribute of a contract cleanly.

2.5.4 Aggregation versus Ternary Relationships *

As we noted in Section 2.4.5, the choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a *relationship set* to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express. For example, consider the ER diagram shown in Figure 2.13. According to this diagram, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is monitored by one or more employees. If we don't need to record the *until* attribute of Monitors, then we might reasonably use a ternary relationship, say, Sponsors2, as shown in Figure 2.20.

Consider the constraint that each sponsorship (of a project by a department) be monitored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship

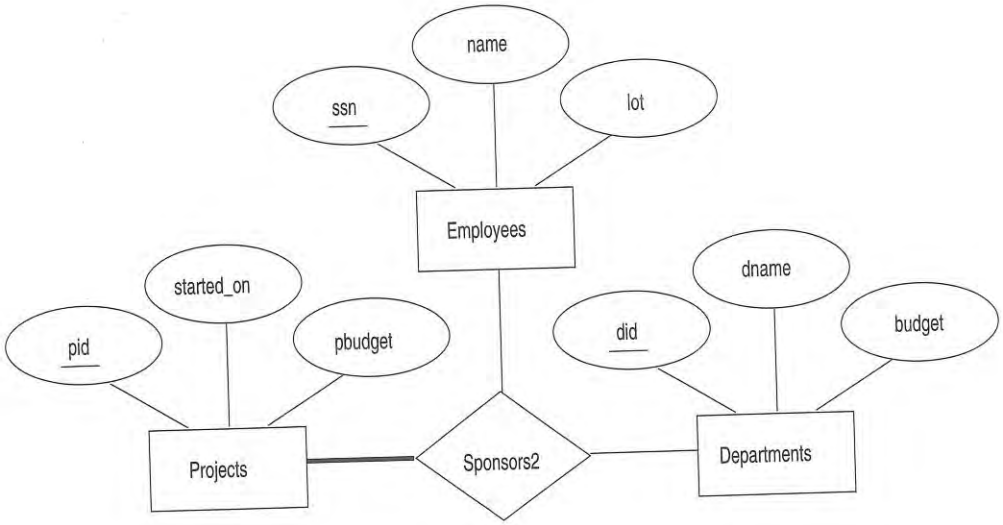


Figure 2.20 Using a Ternary Relationship instead of Aggregation

Monitors in Figure 2.13. Thus, the presence of such a constraint serves as another reason for using aggregation rather than a ternary relationship set.

2.6 CONCEPTUAL DESIGN FOR LARGE ENTERPRISES *

We have thus far concentrated on the constructs available in the ER model for describing various application concepts and relationships. The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams. For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups. Using a high-level, semantic data model such as ER diagrams for conceptual design in such an environment offers the additional advantage that the high-level design can be diagrammatically represented and is easily understood by the many people who must provide input to the design process.

An important aspect of the design process is the methodology used to structure the development of the overall design and to ensure that the design takes into account all user requirements and is consistent. The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, but it allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.

An alternative approach is to develop separate conceptual schemas for different user groups and to then *integrate* these conceptual schemas. To integrate multiple conceptual schemas, we must establish correspondences between entities, relationships, and attributes, and we must resolve numerous kinds of conflicts (e.g., naming conflicts, domain mismatches, differences in measurement units). This task is difficult in its own right. In some situations schema integration cannot be avoided—for example, when one organization merges with another, existing databases may have to be integrated. Schema integration is also increasing in importance as users demand access to *heterogeneous* data sources, often maintained by different organizations.

2.7 POINTS TO REVIEW

- Database design has six steps: requirements analysis, conceptual database design, logical database design, schema refinement, physical database design, and security design. Conceptual design should produce a high-level description of the data, and the entity-relationship (ER) data model provides a graphical approach to this design phase. **(Section 2.1)**
- In the ER model, a real-world object is represented as an *entity*. An *entity set* is a collection of structurally identical entities. Entities are described using *attributes*. Each entity set has a distinguished set of attributes called a *key* that can be used to uniquely identify each entity. **(Section 2.2)**
- A *relationship* is an association between two or more entities. A *relationship set* is a collection of relationships that relate entities from the same entity sets. A relationship can also have *descriptive attributes*. **(Section 2.3)**
- A *key constraint* between an entity set S and a relationship set restricts instances of the relationship set by requiring that each entity of S participate in at most one relationship. A *participation constraint* between an entity set S and a relationship set restricts instances of the relationship set by requiring that each entity of S participate in at least one relationship. The identity and existence of a *weak entity* depends on the identity and existence of another (*owner*) entity. *Class hierarchies* organize structurally similar entities through inheritance into sub- and superclasses. *Aggregation* conceptually transforms a relationship set into an entity set such that the resulting construct can be related to other entity sets. **(Section 2.4)**
- Development of an ER diagram involves important modeling decisions. A thorough understanding of the problem being modeled is necessary to decide whether to use an attribute or an entity set, an entity or a relationship set, a binary or ternary relationship, or aggregation. **(Section 2.5)**
- Conceptual design for large enterprises is especially challenging because data from many sources, managed by many groups, is involved. **(Section 2.6)**

EXERCISES

Exercise 2.1 Explain the following terms briefly: *attribute, domain, entity, relationship, entity set, relationship set, one-to-many relationship, many-to-many relationship, participation constraint, overlap constraint, covering constraint, weak entity set, aggregation*, and *role indicator*.

Exercise 2.2 A university database contains information about professors (identified by social security number, or SSN) and courses (identified by courseid). Professors teach courses; each of the following situations concerns the Teaches relationship set. For each situation, draw an ER diagram that describes it (assuming that no further constraints hold).

1. Professors can teach the same course in several semesters, and each offering must be recorded.
2. Professors can teach the same course in several semesters, and only the most recent such offering needs to be recorded. (Assume this condition applies in all subsequent questions.)
3. Every professor must teach some course.
4. Every professor teaches exactly one course (no more, no less).
5. Every professor teaches exactly one course (no more, no less), and every course must be taught by some professor.
6. Now suppose that certain courses can be taught by a team of professors jointly, but it is possible that no one professor in a team can teach the course. Model this situation, introducing additional entity sets and relationship sets if necessary.

Exercise 2.3 Consider the following information about a university database:

- Professors have an SSN, a name, an age, a rank, and a research specialty.
- Projects have a project number, a sponsor name (e.g., NSF), a starting date, an ending date, and a budget.
- Graduate students have an SSN, a name, an age, and a degree program (e.g., M.S. or Ph.D.).
- Each project is managed by one professor (known as the project's principal investigator).
- Each project is worked on by one or more professors (known as the project's co-investigators).
- Professors can manage and/or work on multiple projects.
- Each project is worked on by one or more graduate students (known as the project's research assistants).
- When graduate students work on a project, a professor must supervise their work on the project. Graduate students can work on multiple projects, in which case they will have a (potentially different) supervisor for each one.
- Departments have a department number, a department name, and a main office.
- Departments have a professor (known as the chairman) who runs the department.
- Professors work in one or more departments, and for each department that they work in, a time percentage is associated with their job.

- Graduate students have one major department in which they are working on their degree.
- Each graduate student has another, more senior graduate student (known as a student advisor) who advises him or her on what courses to take.

Design and draw an ER diagram that captures the information about the university. Use only the basic ER model here, that is, entities, relationships, and attributes. Be sure to indicate any key and participation constraints.

Exercise 2.4 A company database needs to store information about employees (identified by *ssn*, with *salary* and *phone* as attributes); departments (identified by *dno*, with *dname* and *budget* as attributes); and children of employees (with *name* and *age* as attributes). Employees *work* in departments; each department is *managed by* an employee; a child must be identified uniquely by *name* when the parent (who is an employee; assume that only one parent works for the company) is known. We are not interested in information about a child once the parent leaves the company.

Draw an ER diagram that captures this information.

Exercise 2.5 Notown Records has decided to store information about musicians who perform on its albums (as well as other company data) in a database. The company has wisely chosen to hire you as a database designer (at your usual consulting fee of $2,500/day).

- Each musician that records at Notown has an SSN, a name, an address, and a phone number. Poorly paid musicians often share the same address, and no address has more than one phone.
- Each instrument that is used in songs recorded at Notown has a name (e.g., guitar, synthesizer, flute) and a musical key (e.g., C, B-flat, E-flat).
- Each album that is recorded on the Notown label has a title, a copyright date, a format (e.g., CD or MC), and an album identifier.
- Each song recorded at Notown has a title and an author.
- Each musician may play several instruments, and a given instrument may be played by several musicians.
- Each album has a number of songs on it, but no song may appear on more than one album.
- Each song is performed by one or more musicians, and a musician may perform a number of songs.
- Each album has exactly one musician who acts as its producer. A musician may produce several albums, of course.

Design a conceptual schema for Notown and draw an ER diagram for your schema. The following information describes the situation that the Notown database must model. Be sure to indicate all key and cardinality constraints and any assumptions that you make. Identify any constraints that you are unable to capture in the ER diagram and briefly explain why you could not express them.

Exercise 2.6 Computer Sciences Department frequent fliers have been complaining to Dane County Airport officials about the poor organization at the airport. As a result, the officials have decided that all information related to the airport should be organized using a DBMS, and you've been hired to design the database. Your first task is to organize the information about all the airplanes that are stationed and maintained at the airport. The relevant information is as follows:

- Every airplane has a registration number, and each airplane is of a specific model.
- The airport accommodates a number of airplane models, and each model is identified by a model number (e.g., DC-10) and has a capacity and a weight.
- A number of technicians work at the airport. You need to store the name, SSN, address, phone number, and salary of each technician.
- Each technician is an expert on one or more plane model(s), and his or her expertise may overlap with that of other technicians. This information about technicians must also be recorded.
- Traffic controllers must have an annual medical examination. For each traffic controller, you must store the date of the most recent exam.
- All airport employees (including technicians) belong to a union. You must store the union membership number of each employee. You can assume that each employee is uniquely identified by the social security number.
- The airport has a number of tests that are used periodically to ensure that airplanes are still airworthy. Each test has a Federal Aviation Administration (FAA) test number, a name, and a maximum possible score.
- The FAA requires the airport to keep track of each time that a given airplane is tested by a given technician using a given test. For each testing event, the information needed is the date, the number of hours the technician spent doing the test, and the score that the airplane received on the test.

1. Draw an ER diagram for the airport database. Be sure to indicate the various attributes of each entity and relationship set; also specify the key and participation constraints for each relationship set. Specify any necessary overlap and covering constraints as well (in English).
2. The FAA passes a regulation that tests on a plane must be conducted by a technician who is an expert on that model. How would you express this constraint in the ER diagram? If you cannot express it, explain briefly.

Exercise 2.7 The Prescriptions-R-X chain of pharmacies has offered to give you a free lifetime supply of medicines if you design its database. Given the rising cost of health care, you agree. Here's the information that you gather:

- Patients are identified by an SSN, and their names, addresses, and ages must be recorded.
- Doctors are identified by an SSN. For each doctor, the name, specialty, and years of experience must be recorded.
- Each pharmaceutical company is identified by name and has a phone number.

- For each drug, the trade name and formula must be recorded. Each drug is sold by a given pharmaceutical company, and the trade name identifies a drug uniquely from among the products of that company. If a pharmaceutical company is deleted, you need not keep track of its products any longer.
- Each pharmacy has a name, address, and phone number.
- Every patient has a primary physician. Every doctor has at least one patient.
- Each pharmacy sells several drugs and has a price for each. A drug could be sold at several pharmacies, and the price could vary from one pharmacy to another.
- Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors. Each prescription has a date and a quantity associated with it. You can assume that if a doctor prescribes the same drug for the same patient more than once, only the last such prescription needs to be stored.
- Pharmaceutical companies have long-term contracts with pharmacies. A pharmaceutical company can contract with several pharmacies, and a pharmacy can contract with several pharmaceutical companies. For each contract, you have to store a start date, an end date, and the text of the contract.
- Pharmacies appoint a supervisor for each contract. There must always be a supervisor for each contract, but the contract supervisor can change over the lifetime of the contract.

1. Draw an ER diagram that captures the above information. Identify any constraints that are not captured by the ER diagram.
2. How would your design change if each drug must be sold at a fixed price by all pharmacies?
3. How would your design change if the design requirements change as follows: If a doctor prescribes the same drug for the same patient more than once, several such prescriptions may have to be stored.

Exercise 2.8 Although you always wanted to be an artist, you ended up being an expert on databases because you love to cook data and you somehow confused 'data base' with 'data baste.' Your old love is still there, however, so you set up a database company, ArtBase, that builds a product for art galleries. The core of this product is a database with a schema that captures all the information that galleries need to maintain. Galleries keep information about artists, their names (which are unique), birthplaces, age, and style of art. For each piece of artwork, the artist, the year it was made, its unique title, its type of art (e.g., painting, lithograph, sculpture, photograph), and its price must be stored. Pieces of artwork are also classified into groups of various kinds, for example, portraits, still lifes, works by Picasso, or works of the 19th century; a given piece may belong to more than one group. Each group is identified by a name (like those above) that describes the group. Finally, galleries keep information about customers. For each customer, galleries keep their unique name, address, total amount of dollars they have spent in the gallery (very important!), and the artists and groups of art that each customer tends to like.

Draw the ER diagram for the database.

BIBLIOGRAPHIC NOTES

Several books provide a good treatment of conceptual design; these include [52] (which also contains a survey of commercial database design tools) and [641].

The ER model was proposed by Chen [145], and extensions have been proposed in a number of subsequent papers. Generalization and aggregation were introduced in [604]. [330] and [514] contain good surveys of semantic data models. Dynamic and temporal aspects of semantic data models are discussed in [658].

[642] discusses a design methodology based on developing an ER diagram and then translating to the relational model. Markowitz considers referential integrity in the context of ER to relational mapping and discusses the support provided in some commercial systems (as of that date) in [446, 447].

The entity-relationship conference proceedings contain numerous papers on conceptual design, with an emphasis on the ER model, for example, [609].

View integration is discussed in several papers, including [84, 118, 153, 207, 465, 480, 479, 596, 608, 657]. [53] is a survey of several integration approaches.

3

THE RELATIONAL MODEL

> TABLE: An arrangement of words, numbers, or signs, or combinations of them, as in parallel columns, to exhibit a set of facts or relations in a definite, compact, and comprehensive form; a synopsis or scheme.
>
> —Webster's *Dictionary of the English Language*

Codd proposed the relational data model in 1970. At that time most database systems were based on one of two older data models (the hierarchical model and the network model); the relational model revolutionized the database field and largely supplanted these earlier models. Prototype relational database management systems were developed in pioneering research projects at IBM and UC-Berkeley by the mid-70s, and several vendors were offering relational database products shortly thereafter. Today, the relational model is by far the dominant data model and is the foundation for the leading DBMS products, including IBM's DB2 family, Informix, Oracle, Sybase, Microsoft's Access and SQLServer, FoxBase, and Paradox. Relational database systems are ubiquitous in the marketplace and represent a multibillion dollar industry.

The relational model is very simple and elegant; a database is a collection of one or more *relations*, where each relation is a table with rows and columns. This simple tabular representation enables even novice users to understand the contents of a database, and it permits the use of simple, high-level languages to query the data. The major advantages of the relational model over the older data models are its simple data representation and the ease with which even complex queries can be expressed.

This chapter introduces the relational model and covers the following issues:

- How is data represented?
- What kinds of integrity constraints can be expressed?
- How can data be created and modified?
- How can data be manipulated and queried?
- How do we obtain a database design in the relational model?
- How are logical and physical data independence achieved?

SQL: It was the query language of the pioneering System-R relational DBMS developed at IBM. Over the years, SQL has become the most widely used language for creating, manipulating, and querying relational DBMSs. Since many vendors offer SQL products, there is a need for a standard that defines 'official SQL.' The existence of a standard allows users to measure a given vendor's version of SQL for completeness. It also allows users to distinguish SQL features that are specific to one product from those that are standard; an application that relies on non-standard features is less portable.
The first SQL standard was developed in 1986 by the American National Standards Institute (ANSI), and was called SQL-86. There was a minor revision in 1989 called SQL-89, and a major revision in 1992 called SQL-92. The International Standards Organization (ISO) collaborated with ANSI to develop SQL-92. Most commercial DBMSs currently support SQL-92. An exciting development is the imminent approval of SQL:1999, a major extension of SQL-92. While the coverage of SQL in this book is based upon SQL-92, we will cover the main extensions of SQL:1999 as well.

While we concentrate on the underlying concepts, we also introduce the **Data Definition Language (DDL)** features of SQL-92, the standard language for creating, manipulating, and querying data in a relational DBMS. This allows us to ground the discussion firmly in terms of real database systems.

We discuss the concept of a relation in Section 3.1 and show how to create relations using the SQL language. An important component of a data model is the set of constructs it provides for specifying conditions that must be satisfied by the data. Such conditions, called *integrity constraints* (ICs), enable the DBMS to reject operations that might corrupt the data. We present integrity constraints in the relational model in Section 3.2, along with a discussion of SQL support for ICs. We discuss how a DBMS enforces integrity constraints in Section 3.3. In Section 3.4 we turn to the mechanism for accessing and retrieving data from the database, *query languages*, and introduce the querying features of SQL, which we examine in greater detail in a later chapter.

We then discuss the step of converting an ER diagram into a relational database schema in Section 3.5. Finally, we introduce *views*, or tables defined using queries, in Section 3.6. Views can be used to define the external schema for a database and thus provide the support for logical data independence in the relational model.

3.1 INTRODUCTION TO THE RELATIONAL MODEL

The main construct for representing data in the relational model is a **relation**. A relation consists of a **relation schema** and a **relation instance**. The relation instance

is a table, and the relation schema describes the column heads for the table. We first describe the relation schema and then the relation instance. The schema specifies the relation's name, the name of each **field** (or **column**, or **attribute**), and the **domain** of each field. A domain is referred to in a relation schema by the **domain name** and has a set of associated **values**.

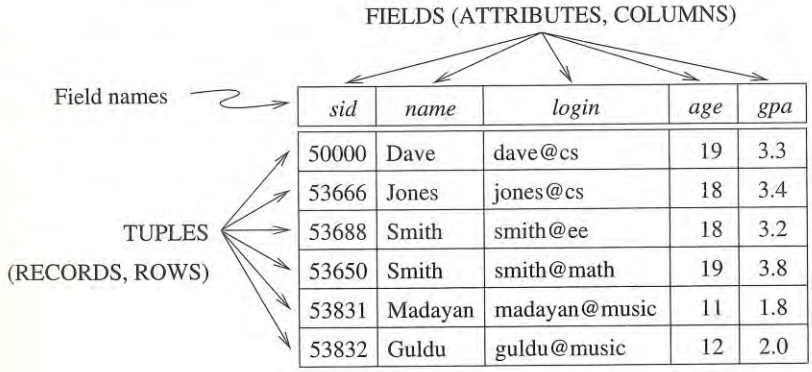
We use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`, *age:* `integer`, *gpa:* `real`)

This says, for instance, that the field named *sid* has a domain named `string`. The set of values associated with domain `string` is the set of all character strings.

We now turn to the instances of a relation. An **instance** of a relation is a set of **tuples**, also called **records**, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a *table* in which each tuple is a *row*, and all rows have the same number of fields. (The term *relation instance* is often abbreviated to just *relation*, when there is no confusion with other aspects of a relation such as its schema.)

An instance of the Students relation appears in Figure 3.1. The instance $S1$ contains



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance $S1$ of the Students Relation

six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. This is a requirement of the relational model—each relation is defined to be a *set* of unique tuples or rows.[1] The order in which the rows are listed is not important. Figure 3.2 shows the same relation instance. If the fields are named, as in

[1] In practice, commercial systems allow tables to have duplicate rows, but we will assume that a relation is indeed a set of tuples unless otherwise noted.

The **degree**, also called **arity**, of a relation is the number of fields. The **cardinality** of a relation instance is the number of tuples in it. In Figure 3.1, the degree of the relation (the number of columns) is five, and the cardinality of this instance is six.

A **relational database** is a collection of relations with distinct relation names. The **relational database schema** is the collection of schemas for the relations in the database. For example, in Chapter 1, we discussed a university database with relations called Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In. An **instance** of a relational database is a collection of relation instances, one per relation schema in the database schema; of course, each relation instance must satisfy the domain constraints in its schema.

3.1.1 Creating and Modifying Relations Using SQL-92

The SQL-92 language standard uses the word *table* to denote *relation*, and we will often follow this convention when discussing SQL. The subset of SQL that supports the creation, deletion, and modification of tables is called the **Data Definition Language (DDL)**. Further, while there is a command that lets users define new domains, analogous to type definition commands in a programming language, we postpone a discussion of domain definition until Section 5.11. For now, we will just consider domains that are built-in types, such as `integer`.

The `CREATE TABLE` statement is used to define a new table.[2] To create the Students relation, we can use the following statement:

```
CREATE TABLE Students ( sid    CHAR(20),
                        name   CHAR(30),
                        login  CHAR(20),
                        age    INTEGER,
                        gpa    REAL )
```

Tuples are inserted using the `INSERT` command. We can insert a single tuple into the Students table as follows:

```
INSERT
INTO   Students   (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
```

We can optionally omit the list of column names in the `INTO` clause and list the values in the appropriate order, but it is good style to be explicit about column names.

[2]SQL also provides statements to destroy tables and to change the columns associated with a table; we discuss these in Section 3.7.

We can delete tuples using the DELETE command. We can delete all Students tuples with *name* equal to Smith using the command:

```
DELETE
FROM    Students S
WHERE   S.name = 'Smith'
```

We can modify the column values in an existing row using the UPDATE command. For example, we can increment the age and decrement the gpa of the student with *sid* 53688:

```
UPDATE  Students S
SET     S.age = S.age + 1, S.gpa = S.gpa - 1
WHERE   S.sid = 53688
```

These examples illustrate some important points. The WHERE clause is applied first and determines which rows are to be modified. The SET clause then determines how these rows are to be modified. If the column that is being modified is also used to determine the new value, the value used in the expression on the right side of equals (=) is the *old* value, that is, before the modification. To illustrate these points further, consider the following variation of the previous query:

```
UPDATE  Students S
SET     S.gpa = S.gpa - 0.1
WHERE   S.gpa >= 3.3
```

If this query is applied on the instance $S1$ of Students shown in Figure 3.1, we obtain the instance shown in Figure 3.3.

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.2
53666	Jones	jones@cs	18	3.3
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.7
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.3 Students Instance $S1$ after Update

3.2 INTEGRITY CONSTRAINTS OVER RELATIONS

A database is only as good as the information stored in it, and a DBMS must therefore help prevent the entry of incorrect information. An **integrity constraint (IC)** is a

condition that is specified on a database schema, and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specified on the database schema, it is a **legal** instance. A DBMS **enforces** integrity constraints, in that it permits only legal instances to be stored in the database.

Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.
2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs. (In some situations, rather than disallow the change, the DBMS might instead make some compensating changes to the data to ensure that the database instance satisfies all ICs. In any case, changes to the database are not allowed to create an instance that violates any IC.)

Many kinds of integrity constraints can be specified in the relational model. We have already seen one example of an integrity constraint in the *domain constraints* associated with a relation schema (Section 3.1). In general, other kinds of constraints can be specified as well; for example, no two students have the same *sid* value. In this section we discuss the integrity constraints, other than domain constraints, that a DBA or user can specify in the relational model.

3.2.1 Key Constraints

Consider the Students relation and the constraint that no two students have the same student id. This IC is an example of a key constraint. A **key constraint** is a statement that a certain *minimal* subset of the fields of a relation is a unique identifier for a tuple. A set of fields that uniquely identifies a tuple according to a key constraint is called a **candidate key** for the relation; we often abbreviate this to just *key*. In the case of the Students relation, the (set of fields containing just the) *sid* field is a candidate key.

Let us take a closer look at the above definition of a (candidate) key. There are two parts to the definition:[3]

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.
2. No subset of the set of fields in a key is a unique identifier for a tuple.

[3]The term *key* is rather overworked. In the context of access methods, we speak of *search keys*, which are quite different.

The first part of the definition means that in *any* legal instance, the values in the key fields uniquely identify a tuple in the instance. When specifying a key constraint, the DBA or user must be sure that this constraint will not prevent them from storing a 'correct' set of tuples. (A similar comment applies to the specification of other kinds of ICs as well.) The notion of 'correctness' here depends upon the nature of the data being stored. For example, several students may have the same name, although each student has a unique student id. If the *name* field is declared to be a key, the DBMS will not allow the Students relation to contain two tuples describing different students with the same name!

The second part of the definition means, for example, that the set of fields {*sid, name*} is not a key for Students, because this set properly contains the key {*sid*}. The set {*sid, name*} is an example of a **superkey**, which is a set of fields that contains a key.

Look again at the instance of the Students relation in Figure 3.1. Observe that two different rows always have different *sid* values; *sid* is a key and uniquely identifies a tuple. However, this does not hold for nonkey fields. For example, the relation contains two rows with *Smith* in the *name* field.

Note that every relation is guaranteed to have a key. Since a relation is a set of tuples, the set of *all* fields is always a superkey. If other constraints hold, some subset of the fields may form a key, but if not, the set of all fields is a key.

A relation may have several candidate keys. For example, the *login* and *age* fields of the Students relation may, taken together, also identify students uniquely. That is, {*login, age*} is also a key. It may seem that *login* is a key, since no two rows in the example instance have the same *login* value. However, the key must identify tuples uniquely in all possible legal instances of the relation. By stating that {*login, age*} is a key, the user is declaring that two students may have the same login or age, but not both.

Out of all the available candidate keys, a database designer can identify a **primary** key. Intuitively, a tuple can be referred to from elsewhere in the database by storing the values of its primary key fields. For example, we can refer to a Students tuple by storing its *sid* value. As a consequence of referring to student tuples in this manner, tuples are frequently accessed by specifying their *sid* value. In principle, we can use any key, not just the primary key, to refer to a tuple. However, using the primary key is preferable because it is what the DBMS expects—this is the significance of designating a particular candidate key as a primary key—and optimizes for. For example, the DBMS may create an index with the primary key fields as the search key, to make the retrieval of a tuple given its primary key value efficient. The idea of referring to a tuple is developed further in the next section.

Specifying Key Constraints in SQL-92

In SQL we can declare that a subset of the columns of a table constitute a key by using the `UNIQUE` constraint. At most one of these 'candidate' keys can be declared to be a *primary key*, using the `PRIMARY KEY` constraint. (SQL does not require that such constraints be declared for a table.)

Let us revisit our example table definition and specify key information:

```
CREATE TABLE Students ( sid    CHAR(20),
                        name   CHAR(30),
                        login  CHAR(20),
                        age    INTEGER,
                        gpa    REAL,
                        UNIQUE (name, age),
                        CONSTRAINT StudentsKey PRIMARY KEY (sid) )
```

This definition says that *sid* is the primary key and that the combination of *name* and *age* is also a key. The definition of the primary key also illustrates how we can name a constraint by preceding it with `CONSTRAINT` *constraint-name*. If the constraint is violated, the constraint name is returned and can be used to identify the error.

3.2.2 Foreign Key Constraints

Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, and perhaps modified, to keep the data consistent. An IC involving both relations must be specified if a DBMS is to make such checks. The most common IC involving two relations is a *foreign key* constraint.

Suppose that in addition to Students, we have a second relation:

Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`)

To ensure that only bona fide students can enroll in courses, any value that appears in the *sid* field of an instance of the Enrolled relation should also appear in the *sid* field of some tuple in the Students relation. The *sid* field of Enrolled is called a **foreign key** and **refers** to Students. The foreign key in the referencing relation (Enrolled, in our example) must match the primary key of the referenced relation (Students), i.e., it must have the same number of columns and compatible data types, although the column names can be different.

This constraint is illustrated in Figure 3.4. As the figure shows, there may well be some students who are not referenced from Enrolled (e.g., the student with *sid=50000*).

However, every *sid* value that appears in the instance of the Enrolled table appears in the primary key column of a row in the Students table.

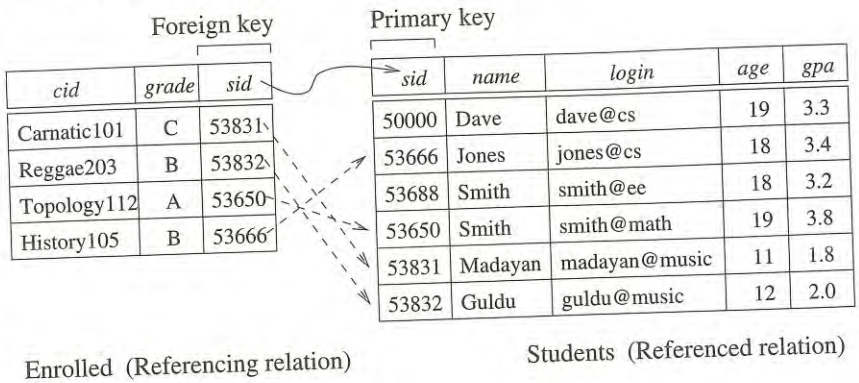


Figure 3.4 Referential Integrity

If we try to insert the tuple $\langle$*55555, Art104, A*$\rangle$ into $E1$, the IC is violated because there is no tuple in $S1$ with the id 55555; the database system should reject such an insertion. Similarly, if we delete the tuple $\langle$*53666, Jones, jones@cs, 18, 3.4*$\rangle$ from $S1$, we violate the foreign key constraint because the tuple $\langle$*53666, History105, B*$\rangle$ in $E1$ contains *sid* value 53666, the *sid* of the deleted Students tuple. The DBMS should disallow the deletion or, perhaps, also delete the Enrolled tuple that refers to the deleted Students tuple. We discuss foreign key constraints and their impact on updates in Section 3.3.

Finally, we note that a foreign key could refer to the same relation. For example, we could extend the Students relation with a column called *partner* and declare this column to be a foreign key referring to Students. Intuitively, every student could then have a partner, and the *partner* field contains the partner's *sid*. The observant reader will no doubt ask, "What if a student does not (yet) have a partner?" This situation is handled in SQL by using a special value called **null**. The use of *null* in a field of a tuple means that value in that field is either unknown or not applicable (e.g., we do not know the partner yet, or there is no partner). The appearance of *null* in a foreign key field does not violate the foreign key constraint. However, *null* values are not allowed to appear in a primary key field (because the primary key fields are used to identify a tuple uniquely). We will discuss *null* values further in Chapter 5.

Specifying Foreign Key Constraints in SQL-92

Let us define Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`):

```
CREATE TABLE Enrolled ( sid    CHAR(20),
```

```
cid    CHAR(20),
grade  CHAR(10),
PRIMARY KEY (sid, cid),
FOREIGN KEY (sid) REFERENCES Students )
```

The foreign key constraint states that every *sid* value in Enrolled must also appear in Students, that is, *sid* in Enrolled is a foreign key referencing Students. Incidentally, the primary key constraint states that a student has exactly one grade for each course that he or she is enrolled in. If we want to record more than one grade per student per course, we should change the primary key constraint.

3.2.3 General Constraints

Domain, primary key, and foreign key constraints are considered to be a fundamental part of the relational data model and are given special attention in most commercial systems. Sometimes, however, it is necessary to specify more general constraints.

For example, we may require that student ages be within a certain range of values; given such an IC specification, the DBMS will reject inserts and updates that violate the constraint. This is very useful in preventing data entry errors. If we specify that all students must be at least 16 years old, the instance of Students shown in Figure 3.1 is illegal because two students are underage. If we disallow the insertion of these two tuples, we have a legal instance, as shown in Figure 3.5.

sid	*name*	*login*	*age*	*gpa*
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8

Figure 3.5 An Instance $S2$ of the Students Relation

The IC that students must be older than 16 can be thought of as an extended domain constraint, since we are essentially defining the set of permissible *age* values more stringently than is possible by simply using a standard domain such as `integer`. In general, however, constraints that go well beyond domain, key, or foreign key constraints can be specified. For example, we could require that every student whose age is greater than 18 must have a gpa greater than 3.

Current relational database systems support such general constraints in the form of *table constraints* and *assertions*. Table constraints are associated with a single table and are checked whenever that table is modified. In contrast, assertions involve several

tables and are checked whenever any of these tables is modified. Both table constraints and assertions can use the full power of SQL queries to specify the desired restriction. We discuss SQL support for *table constraints* and *assertions* in Section 5.11 because a full appreciation of their power requires a good grasp of SQL's query capabilities.

3.3 ENFORCING INTEGRITY CONSTRAINTS

As we observed earlier, ICs are specified when a relation is created and enforced when a relation is modified. The impact of domain, `PRIMARY KEY`, and `UNIQUE` constraints is straightforward: if an insert, delete, or update command causes a violation, it is rejected. Potential IC violation is generally checked at the end of each SQL statement execution, although it can be *deferred* until the end of the transaction executing the statement, as we will see in Chapter 18.

Consider the instance $S1$ of Students shown in Figure 3.1. The following insertion violates the primary key constraint because there is already a tuple with the *sid* 53688, and it will be rejected by the DBMS:

```
INSERT
INTO   Students   (sid, name, login, age, gpa)
VALUES (53688, 'Mike', 'mike@ee', 17, 3.4)
```

The following insertion violates the constraint that the primary key cannot contain *null*:

```
INSERT
INTO   Students   (sid, name, login, age, gpa)
VALUES (null, 'Mike', 'mike@ee', 17, 3.4)
```

Of course, a similar problem arises whenever we try to insert a tuple with a value in a field that is not in the domain associated with that field, i.e., whenever we violate a domain constraint. Deletion does not cause a violation of domain, primary key or unique constraints. However, an update can cause violations, similar to an insertion:

```
UPDATE Students S
SET    S.sid = 50000
WHERE  S.sid = 53688
```

This update violates the primary key constraint because there is already a tuple with *sid* 50000.

The impact of foreign key constraints is more complex because SQL sometimes tries to rectify a foreign key constraint violation instead of simply rejecting the change. We will

discuss the **referential integrity enforcement steps** taken by the DBMS in terms of our Enrolled and Students tables, with the foreign key constraint that Enrolled.*sid* is a reference to (the primary key of) Students.

In addition to the instance $S1$ of Students, consider the instance of Enrolled shown in Figure 3.4. Deletions of Enrolled tuples do not violate referential integrity, but insertions of Enrolled tuples could. The following insertion is illegal because there is no student with *sid* 51111:

```
INSERT
INTO   Enrolled   (cid, grade, sid)
VALUES ('Hindi101', 'B', 51111)
```

On the other hand, insertions of Students tuples do not violate referential integrity although deletions could. Further, updates on either Enrolled or Students that change the *sid* value could potentially violate referential integrity.

SQL-92 provides several alternative ways to handle foreign key violations. We must consider three basic questions:

1. *What should we do if an Enrolled row is inserted, with a* sid *column value that does not appear in any row of the Students table?*

   In this case the `INSERT` command is simply rejected.

2. *What should we do if a Students row is deleted?*

   The options are:

   - Delete all Enrolled rows that refer to the deleted Students row.
   - Disallow the deletion of the Students row if an Enrolled row refers to it.
   - Set the *sid* column to the *sid* of some (existing) 'default' student, for every Enrolled row that refers to the deleted Students row.
   - For every Enrolled row that refers to it, set the *sid* column to *null*. In our example, this option conflicts with the fact that *sid* is part of the primary key of Enrolled and therefore cannot be set to *null*. Thus, we are limited to the first three options in our example, although this fourth option (setting the foreign key to *null*) is available in the general case.

3. *What should we do if the primary key value of a Students row is updated?*

   The options here are similar to the previous case.

SQL-92 allows us to choose any of the four options on `DELETE` and `UPDATE`. For example, we can specify that when a Students row is *deleted*, all Enrolled rows that refer to it are to be deleted as well, but that when the *sid* column of a Students row is *modified*, this update is to be rejected if an Enrolled row refers to the modified Students row:

```
CREATE TABLE Enrolled ( sid    CHAR(20),
                        cid    CHAR(20),
                        grade  CHAR(10),
                        PRIMARY KEY (sid, cid),
                        FOREIGN KEY (sid) REFERENCES Students,
                                    ON DELETE CASCADE,
                                    ON UPDATE NO ACTION )
```

The options are specified as part of the foreign key declaration. The default option is NO ACTION, which means that the action (DELETE or UPDATE) is to be rejected. Thus, the ON UPDATE clause in our example could be omitted, with the same effect. The CASCADE keyword says that if a Students row is deleted, all Enrolled rows that refer to it are to be deleted as well. If the UPDATE clause specified CASCADE, and the *sid* column of a Students row is updated, this update is also carried out in each Enrolled row that refers to the updated Students row.

If a Students row is deleted, we can switch the enrollment to a 'default' student by using ON DELETE SET DEFAULT. The default student is specified as part of the definition of the *sid* field in Enrolled; for example, *sid* CHAR(20) DEFAULT *'53666'*. Although the specification of a default value is appropriate in some situations (e.g., a default parts supplier if a particular supplier goes out of business), it is really not appropriate to switch enrollments to a default student. The correct solution in this example is to also delete all enrollment tuples for the deleted student (that is, CASCADE), or to reject the update.

SQL also allows the use of *null* as the default value by specifying ON DELETE SET NULL.

3.4 QUERYING RELATIONAL DATA

A **relational database query** (query, for short) is a question about the data, and the answer consists of a new relation containing the result. For example, we might want to find all students younger than 18 or all students enrolled in Reggae203. A **query language** is a specialized language for writing queries.

SQL is the most popular commercial query language for a relational DBMS. We now present some SQL examples that illustrate how easily relations can be queried. Consider the instance of the Students relation shown in Figure 3.1. We can retrieve rows corresponding to students who are younger than 18 with the following SQL query:

```
SELECT *
FROM   Students S
WHERE  S.age < 18
```

The symbol * means that we retain all fields of selected tuples in the result. To understand this query, think of S as a variable that takes on the value of each tuple in Students, one tuple after the other. The condition *S.age* < *18* in the WHERE clause specifies that we want to select only tuples in which the *age* field has a value less than 18. This query evaluates to the relation shown in Figure 3.6.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.6 Students with *age* < 18 on Instance *S1*

This example illustrates that the domain of a field restricts the operations that are permitted on field values, in addition to restricting the values that can appear in the field. The condition *S.age* < *18* involves an arithmetic comparison of an *age* value with an integer and is permissible because the domain of *age* is the set of integers. On the other hand, a condition such as *S.age* = *S.sid* does not make sense because it compares an integer value with a string value, and this comparison is defined to fail in SQL; a query containing this condition will produce no answer tuples.

In addition to selecting a subset of tuples, a query can extract a subset of the fields of each selected tuple. We can compute the names and logins of students who are younger than 18 with the following query:

```
SELECT S.name, S.login
FROM   Students S
WHERE  S.age < 18
```

Figure 3.7 shows the answer to this query; it is obtained by applying the selection to the instance *S1* of Students (to get the relation shown in Figure 3.6), followed by removing unwanted fields. Note that the order in which we perform these operations does matter—if we remove unwanted fields first, we cannot check the condition *S.age* < *18*, which involves one of those fields.

We can also combine information in the Students and Enrolled relations. If we want to obtain the names of all students who obtained an A and the id of the course in which they got an A, we could write the following query:

```
SELECT S.name, E.cid
FROM   Students S, Enrolled E
WHERE  S.sid = E.sid AND E.grade = 'A'
```

DISTINCT types in SQL: A comparison of two values drawn from different domains should fail, even if the values are 'compatible' in the sense that both are numeric or both are string values etc. For example, if *salary* and *age* are two different domains whose values are represented as integers, a comparison of a salary value with an age value should fail. Unfortunately, SQL-92's support for the concept of domains does not go this far: We are forced to define *salary* and *age* as integer types and the comparison $S < A$ will succeed when S is bound to the salary value 25 and A is bound to the age value 50. The latest version of the SQL standard, called SQL:1999, addresses this problem, and allows us to define *salary* and *age* as DISTINCT types even though their values are *represented* as integers. Many systems, e.g., Informix UDS and IBM DB2, already support this feature.

name	*login*
Madayan	madayan@music
Guldu	guldu@music

Figure 3.7 Names and Logins of Students under 18

This query can be understood as follows: "If there is a Students tuple S and an Enrolled tuple E such that S.sid = E.sid (so that S describes the student who is enrolled in E) and E.grade = 'A', then print the student's name and the course id." When evaluated on the instances of Students and Enrolled in Figure 3.4, this query returns a single tuple, $\langle$*Smith, Topology112*$\rangle$.

We will cover relational queries, and SQL in particular, in more detail in subsequent chapters.

3.5 LOGICAL DATABASE DESIGN: ER TO RELATIONAL

The ER model is convenient for representing an initial, high-level database design. Given an ER diagram describing a database, there is a standard approach to generating a relational database schema that closely approximates the ER design. (The translation is approximate to the extent that we cannot capture all the constraints implicit in the ER design using SQL-92, unless we use certain SQL-92 constraints that are costly to check.) We now describe how to translate an ER diagram into a collection of tables with associated constraints, i.e., a relational database schema.

3.5.1 Entity Sets to Tables

An entity set is mapped to a relation in a straightforward way: Each attribute of the entity set becomes an attribute of the table. Note that we know both the domain of each attribute and the (primary) key of an entity set.

Consider the Employees entity set with attributes *ssn*, *name*, and *lot* shown in Figure 3.8. A possible instance of the Employees entity set, containing three Employees

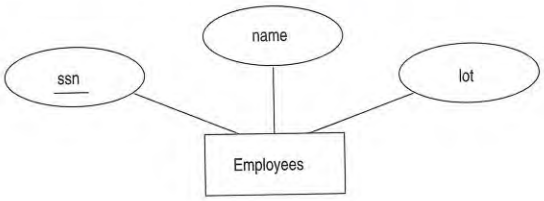


Figure 3.8 The Employees Entity Set

entities, is shown in Figure 3.9 in a tabular format.

ssn	*name*	*lot*
123-22-3666	Attishoo	48
231-31-5368	Smiley	22
131-24-3650	Smethurst	35

Figure 3.9 An Instance of the Employees Entity Set

The following SQL statement captures the preceding information, including the domain constraints and key information:

```
CREATE TABLE Employees ( ssn      CHAR(11),
                         name     CHAR(30),
                         lot      INTEGER,
                         PRIMARY KEY (ssn) )
```

3.5.2 Relationship Sets (without Constraints) to Tables

A relationship set, like an entity set, is mapped to a relation in the relational model. We begin by considering relationship sets without key and participation constraints, and we discuss how to handle such constraints in subsequent sections. To represent a relationship, we must be able to identify each participating entity and give values

to the descriptive attributes of the relationship. Thus, the attributes of the relation include:

- The primary key attributes of each participating entity set, as foreign key fields.
- The descriptive attributes of the relationship set.

The set of nondescriptive attributes is a superkey for the relation. If there are no key constraints (see Section 2.4.1), this set of attributes is a candidate key.

Consider the Works_In2 relationship set shown in Figure 3.10. Each department has offices in several locations and we want to record the locations at which each employee works.

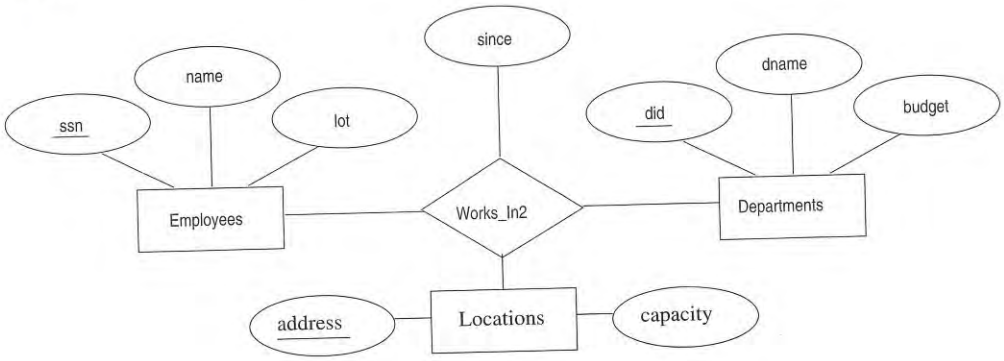


Figure 3.10 A Ternary Relationship Set

All the available information about the Works_In2 table is captured by the following SQL definition:

```
CREATE TABLE Works_In2 ( ssn      CHAR(11),
                         did      INTEGER,
                         address  CHAR(20),
                         since    DATE,
                         PRIMARY KEY (ssn, did, address),
                         FOREIGN KEY (ssn) REFERENCES Employees,
                         FOREIGN KEY (address) REFERENCES Locations,
                         FOREIGN KEY (did) REFERENCES Departments )
```

Note that the *address*, *did*, and *ssn* fields cannot take on *null* values. Because these fields are part of the primary key for Works_In2, a `NOT NULL` constraint is implicit for each of these fields. This constraint ensures that these fields uniquely identify a department, an employee, and a location in each tuple of Works_In. We can also

specify that a particular action is desired when a referenced Employees, Departments or Locations tuple is deleted, as explained in the discussion of integrity constraints in Section 3.2. In this chapter we assume that the default action is appropriate except for situations in which the semantics of the ER diagram require some other action.

Finally, consider the Reports_To relationship set shown in Figure 3.11. The role in-

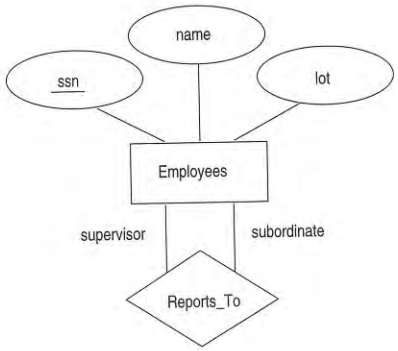


Figure 3.11 The Reports_To Relationship Set

dicators *supervisor* and *subordinate* are used to create meaningful field names in the CREATE statement for the Reports_To table:

```
CREATE TABLE Reports_To (
        supervisor_ssn   CHAR(11),
        subordinate_ssn  CHAR(11),
        PRIMARY KEY (supervisor_ssn, subordinate_ssn),
        FOREIGN KEY (supervisor_ssn) REFERENCES Employees(ssn),
        FOREIGN KEY (subordinate_ssn) REFERENCES Employees(ssn) )
```

Observe that we need to explicitly name the referenced field of Employees because the field name differs from the name(s) of the referring field(s).

3.5.3 Translating Relationship Sets with Key Constraints

If a relationship set involves n entity sets and some m of them are linked via arrows in the ER diagram, the key for any one of these m entity sets constitutes a key for the relation to which the relationship set is mapped. Thus we have m candidate keys, and one of these should be designated as the primary key. The translation discussed in Section 2.3 from relationship sets to a relation can be used in the presence of key constraints, taking into account this point about keys.

Consider the relationship set Manages shown in Figure 3.12. The table corresponding

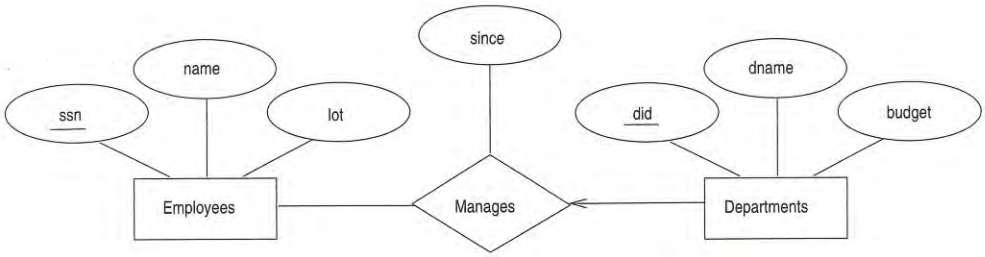


Figure 3.12 Key Constraint on Manages

to Manages has the attributes *ssn, did, since.* However, because each department has at most one manager, no two tuples can have the same *did* value but differ on the *ssn* value. A consequence of this observation is that *did* is itself a key for Manages; indeed, the set *did, ssn* is not a key (because it is not minimal). The Manages relation can be defined using the following SQL statement:

```
CREATE TABLE Manages ( ssn       CHAR(11),
                       did       INTEGER,
                       since     DATE,
                       PRIMARY KEY (did),
                       FOREIGN KEY (ssn) REFERENCES Employees,
                       FOREIGN KEY (did) REFERENCES Departments )
```

A second approach to translating a relationship set with key constraints is often superior because it avoids creating a distinct table for the relationship set. The idea is to include the information about the relationship set in the table corresponding to the entity set with the key, taking advantage of the key constraint. In the Manages example, because a department has at most one manager, we can add the key fields of the Employees tuple denoting the manager and the *since* attribute to the Departments tuple.

This approach eliminates the need for a separate Manages relation, and queries asking for a department's manager can be answered without combining information from two relations. The only drawback to this approach is that space could be wasted if several departments have no managers. In this case the added fields would have to be filled with *null* values. The first translation (using a separate table for Manages) avoids this inefficiency, but some important queries require us to combine information from two relations, which can be a slow operation.

The following SQL statement, defining a Dept_Mgr relation that captures the information in both Departments and Manages, illustrates the second approach to translating relationship sets with key constraints:

```
CREATE TABLE Dept_Mgr ( did      INTEGER,
                        dname    CHAR(20),
                        budget   REAL,
                        ssn      CHAR(11),
                        since    DATE,
                        PRIMARY KEY (did),
                        FOREIGN KEY (ssn) REFERENCES Employees )
```

Note that *ssn* can take on *null* values.

This idea can be extended to deal with relationship sets involving more than two entity sets. In general, if a relationship set involves n entity sets and some m of them are linked via arrows in the ER diagram, the relation corresponding to any one of the m sets can be augmented to capture the relationship.

We discuss the relative merits of the two translation approaches further after considering how to translate relationship sets with participation constraints into tables.

3.5.4 Translating Relationship Sets with Participation Constraints

Consider the ER diagram in Figure 3.13, which shows two relationship sets, Manages and Works_In.

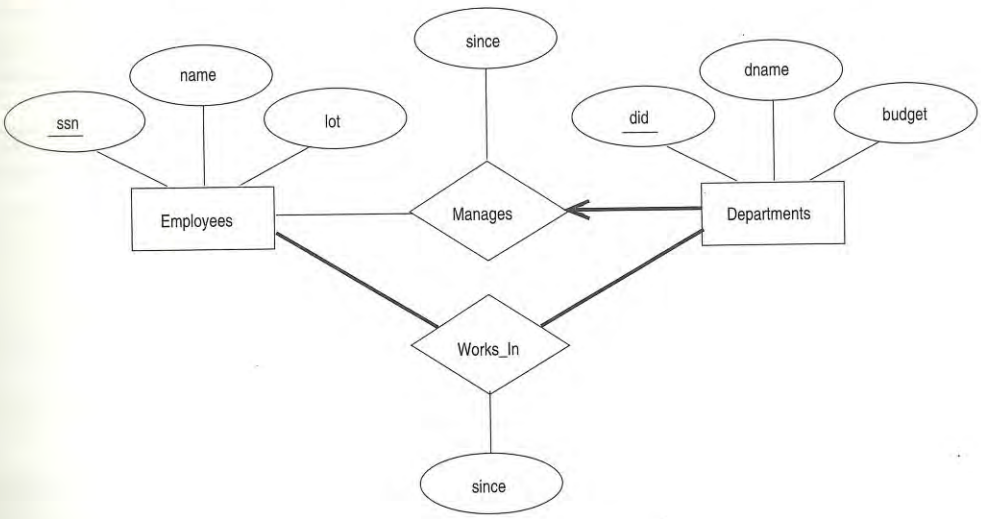


Figure 3.13 Manages and Works_In

Every department is required to have a manager, due to the participation constraint, and at most one manager, due to the key constraint. The following SQL statement reflects the second translation approach discussed in Section 3.5.3, and uses the key constraint:

```
CREATE TABLE Dept_Mgr ( did      INTEGER,
                        dname    CHAR(20),
                        budget   REAL,
                        ssn      CHAR(11) NOT NULL,
                        since    DATE,
                        PRIMARY KEY (did),
                        FOREIGN KEY (ssn) REFERENCES Employees,
                        ON DELETE NO ACTION )
```

It also captures the participation constraint that every department must have a manager: Because *ssn* cannot take on *null* values, each tuple of Dept_Mgr identifies a tuple in Employees (who is the manager). The `NO ACTION` specification, which is the default and need not be explicitly specified, ensures that an Employees tuple cannot be deleted while it is pointed to by a Dept_Mgr tuple. If we wish to delete such an Employees tuple, we must first change the Dept_Mgr tuple to have a new employee as manager. (We could have specified `CASCADE` instead of `NO ACTION`, but deleting all information about a department just because its manager has been fired seems a bit extreme!)

The constraint that every department must have a manager cannot be captured using the first translation approach discussed in Section 3.5.3. (Look at the definition of Manages and think about what effect it would have if we added `NOT NULL` constraints to the *ssn* and *did* fields. *Hint:* The constraint would prevent the firing of a manager, but does not ensure that a manager is initially appointed for each department!) This situation is a strong argument in favor of using the second approach for one-to-many relationships such as Manages, especially when the entity set with the key constraint also has a total participation constraint.

Unfortunately, there are many participation constraints that we cannot capture using SQL-92, short of using *table constraints* or *assertions*. Table constraints and assertions can be specified using the full power of the SQL query language (as discussed in Section 5.11) and are very expressive, but also very expensive to check and enforce. For example, we cannot enforce the participation constraints on the Works_In relation without using these general constraints. To see why, consider the Works_In relation obtained by translating the ER diagram into relations. It contains fields *ssn* and *did*, which are foreign keys referring to Employees and Departments. To ensure total participation of Departments in Works_In, we have to guarantee that every *did* value in Departments appears in a tuple of Works_In. We could try to guarantee this condition by declaring that *did* in Departments is a foreign key referring to Works_In, but this is not a valid foreign key constraint because *did* is not a candidate key for Works_In.

To ensure total participation of Departments in Works_In using SQL-92, we need an assertion. We have to guarantee that every *did* value in Departments appears in a tuple of Works_In; further, this tuple of Works_In must also have non *null* values in the fields that are foreign keys referencing other entity sets involved in the relationship (in this example, the *ssn* field). We can ensure the second part of this constraint by imposing the stronger requirement that *ssn* in Works_In cannot contain *null* values. (Ensuring that the participation of Employees in Works_In is total is symmetric.)

Another constraint that requires assertions to express in SQL is the requirement that each Employees entity (in the context of the Manages relationship set) must manage at least one department.

In fact, the Manages relationship set exemplifies most of the participation constraints that we can capture using key and foreign key constraints. Manages is a binary relationship set in which exactly one of the entity sets (Departments) has a key constraint, and the total participation constraint is expressed on that entity set.

We can also capture participation constraints using key and foreign key constraints in one other special situation: a relationship set in which all participating entity sets have key constraints and total participation. The best translation approach in this case is to map all the entities as well as the relationship into a single table; the details are straightforward.

3.5.5 Translating Weak Entity Sets

A weak entity set always participates in a one-to-many binary relationship and has a key constraint and total participation. The second translation approach discussed in Section 3.5.3 is ideal in this case, but we must take into account the fact that the weak entity has only a partial key. Also, when an owner entity is deleted, we want all owned weak entities to be deleted.

Consider the Dependents weak entity set shown in Figure 3.14, with partial key *pname*. A Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity, and the Dependents entity must be deleted if the owning Employees entity is deleted.

We can capture the desired semantics with the following definition of the Dep_Policy relation:

```
CREATE TABLE Dep_Policy ( pname   CHAR(20),
                          age     INTEGER,
                          cost    REAL,
                          ssn     CHAR(11),
```

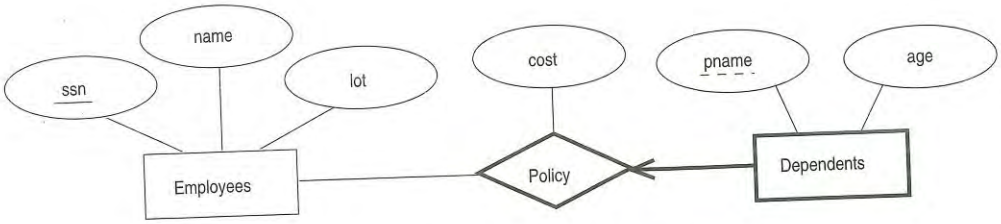


Figure 3.14 The Dependents Weak Entity Set

```
PRIMARY KEY (pname, ssn),
FOREIGN KEY (ssn) REFERENCES Employees,
ON DELETE CASCADE )
```

Observe that the primary key is $\langle pname,\ ssn \rangle$, since Dependents is a weak entity. This constraint is a change with respect to the translation discussed in Section 3.5.3. We have to ensure that every Dependents entity is associated with an Employees entity (the owner), as per the total participation constraint on Dependents. That is, *ssn* cannot be *null*. This is ensured because *ssn* is part of the primary key. The CASCADE option ensures that information about an employee's policy and dependents is deleted if the corresponding Employees tuple is deleted.

3.5.6 Translating Class Hierarchies

We present the two basic approaches to handling ISA hierarchies by applying them to the ER diagram shown in Figure 3.15:

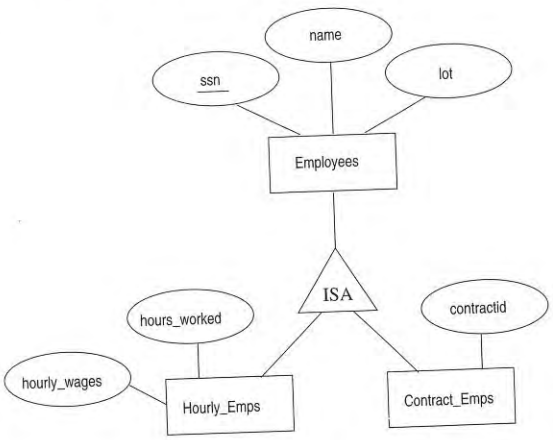


Figure 3.15 Class Hierarchy

1. We can map each of the entity sets Employees, Hourly_Emps, and Contract_Emps to a distinct relation. The Employees relation is created as in Section 2.2. We discuss Hourly_Emps here; Contract_Emps is handled similarly. The relation for Hourly_Emps includes the *hourly_wages* and *hours_worked* attributes of Hourly_Emps. It also contains the key attributes of the superclass (*ssn*, in this example), which serve as the primary key for Hourly_Emps, as well as a foreign key referencing the superclass (Employees). For each Hourly_Emps entity, the value of the *name* and *lot* attributes are stored in the corresponding row of the superclass (Employees). Note that if the superclass tuple is deleted, the delete must be cascaded to Hourly_Emps.

2. Alternatively, we can create just two relations, corresponding to Hourly_Emps and Contract_Emps. The relation for Hourly_Emps includes all the attributes of Hourly_Emps as well as all the attributes of Employees (i.e., *ssn, name, lot, hourly_wages, hours_worked*).

The first approach is general and is always applicable. Queries in which we want to examine all employees and do not care about the attributes specific to the subclasses are handled easily using the Employees relation. However, queries in which we want to examine, say, hourly employees, may require us to combine Hourly_Emps (or Contract_Emps, as the case may be) with Employees to retrieve *name* and *lot*.

The second approach is not applicable if we have employees who are neither hourly employees nor contract employees, since there is no way to store such employees. Also, if an employee is both an Hourly_Emps and a Contract_Emps entity, then the *name* and *lot* values are stored twice. This duplication can lead to some of the anomalies that we discuss in Chapter 15. A query that needs to examine all employees must now examine two relations. On the other hand, a query that needs to examine only hourly employees can now do so by examining just one relation. The choice between these approaches clearly depends on the semantics of the data and the frequency of common operations.

In general, overlap and covering constraints can be expressed in SQL-92 only by using assertions.

3.5.7 Translating ER Diagrams with Aggregation

Translating aggregation into the relational model is easy because there is no real distinction between entities and relationships in the relational model.

Consider the ER diagram shown in Figure 3.16. The Employees, Projects, and Departments entity sets and the Sponsors relationship set are mapped as described in previous sections. For the Monitors relationship set, we create a relation with the following attributes: the key attributes of Employees (*ssn*), the key attributes of Spon-

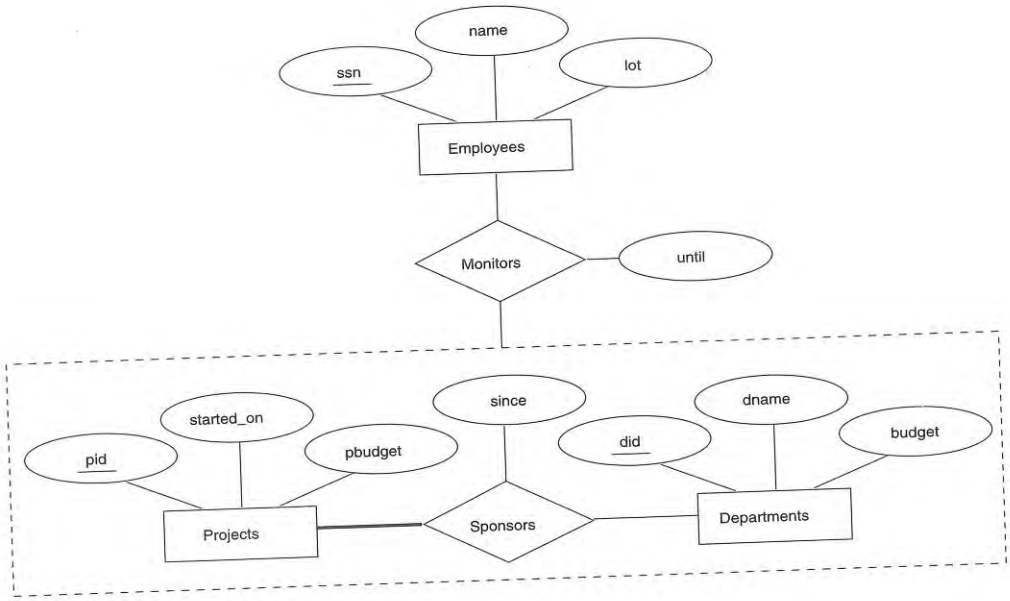


Figure 3.16 Aggregation

sors (*did, pid*), and the descriptive attributes of Monitors (*until*). This translation is essentially the standard mapping for a relationship set, as described in Section 3.5.2.

There is a special case in which this translation can be refined further by dropping the Sponsors relation. Consider the Sponsors relation. It has attributes *pid, did,* and *since*, and in general we need it (in addition to Monitors) for two reasons:

1. We have to record the descriptive attributes (in our example, *since*) of the Sponsors relationship.
2. Not every sponsorship has a monitor, and thus some ⟨*pid, did*⟩ pairs in the Sponsors relation may not appear in the Monitors relation.

However, if Sponsors has no descriptive attributes and has total participation in Monitors, every possible instance of the Sponsors relation can be obtained by looking at the ⟨*pid, did*⟩ columns of the Monitors relation. Thus, we need not store the Sponsors relation in this case.

3.5.8 ER to Relational: Additional Examples *

Consider the ER diagram shown in Figure 3.17. We can translate this ER diagram into the relational model as follows, taking advantage of the key constraints to combine Purchaser information with Policies and Beneficiary information with Dependents:

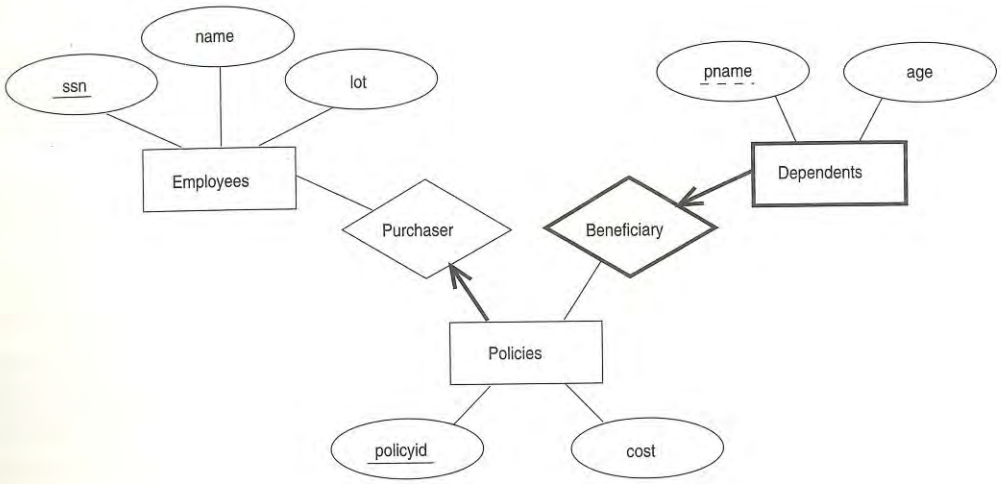


Figure 3.17 Policy Revisited

```
CREATE TABLE Policies ( policyid INTEGER,
                        cost     REAL,
                        ssn      CHAR(11) NOT NULL,
                        PRIMARY KEY (policyid),
                        FOREIGN KEY (ssn) REFERENCES Employees,
                        ON DELETE CASCADE)
```

```
CREATE TABLE Dependents ( pname   CHAR(20),
                          age     INTEGER,
                          policyid INTEGER,
                          PRIMARY KEY (pname, policyid),
                          FOREIGN KEY (policyid) REFERENCES Policies,
                          ON DELETE CASCADE )
```

Notice how the deletion of an employee leads to the deletion of all policies owned by the employee and all dependents who are beneficiaries of those policies. Further, each dependent is required to have a covering policy—because *policyid* is part of the primary key of Dependents, there is an implicit `NOT NULL` constraint. This model accurately reflects the participation constraints in the ER diagram and the intended actions when an employee entity is deleted.

In general, there could be a chain of identifying relationships for weak entity sets. For example, we assumed that *policyid* uniquely identifies a policy. Suppose that *policyid* only distinguishes the policies owned by a given employee; that is, *policyid* is only a partial key and Policies should be modeled as a weak entity set. This new assumption

about *policyid* does not cause much to change in the preceding discussion. In fact, the only changes are that the primary key of Policies becomes ⟨*policyid, ssn*⟩, and as a consequence, the definition of Dependents changes—a field called *ssn* is added and becomes part of both the primary key of Dependents and the foreign key referencing Policies:

```
CREATE TABLE Dependents ( pname    CHAR(20),
                          ssn      CHAR(11),
                          age      INTEGER,
                          policyid INTEGER NOT NULL,
                          PRIMARY KEY (pname, policyid, ssn),
                          FOREIGN KEY (policyid, ssn) REFERENCES Policies,
                          ON DELETE CASCADE)
```

3.6 INTRODUCTION TO VIEWS

A **view** is a table whose rows are not explicitly stored in the database but are computed as needed from a **view definition**. Consider the Students and Enrolled relations. Suppose that we are often interested in finding the names and logins of students who got a grade of B in some course, together with the cid for the course. We can define a view for this purpose. Using SQL-92 notation:

```
CREATE VIEW B-Students (name, login, course)
       AS SELECT S.sname, S.sid, E.cid
          FROM   Students S, Enrolled E
          WHERE  S.sid = E.sid AND E.grade = 'B'
```

The view B-Students has three fields called *name*, *sid*, and *course* with the same domains as the fields *sname* and *sid* in *Students* and *cid* in Enrolled. (If the optional arguments *name*, *sid*, and *course* are omitted from the CREATE VIEW statement, the column names *sname*, *sid*, and *cid* are inherited.)

This view can be used just like a **base table**, or explicitly stored table, in defining new queries or views. Given the instances of Enrolled and Students shown in Figure 3.4, B-Students contains the tuples shown in Figure 3.18. Conceptually, whenever B-Students is used in a query, the view definition is first evaluated to obtain the corresponding instance of B-Students, and then the rest of the query is evaluated treating B-Students like any other relation referred to in the query. (We will discuss how queries on views are evaluated in practice in Chapter 23.)

name	*sid*	*course*
Jones	53666	History105
Guldu	53832	Reggae203

Figure 3.18 An Instance of the B-Students View

3.6.1 Views, Data Independence, Security

Consider the levels of abstraction that we discussed in Section 1.5.2. The *physical* schema for a relational database describes how the relations in the conceptual schema are stored, in terms of the file organizations and indexes used. The *conceptual* schema is the collection of schemas of the relations stored in the database. While some relations in the conceptual schema can also be exposed to applications, i.e., be part of the *external* schema of the database, additional relations in the *external* schema can be defined using the view mechanism. The view mechanism thus provides the support for *logical data independence* in the relational model. That is, it can be used to define relations in the external schema that mask changes in the conceptual schema of the database from applications. For example, if the schema of a stored relation is changed, we can define a view with the old schema, and applications that expect to see the old schema can now use this view.

Views are also valuable in the context of *security*: We can define views that give a group of users access to just the information they are allowed to see. For example, we can define a view that allows students to see other students' name and age but not their gpa, and allow all students to access this view, but not the underlying Students table (see Chapter 17).

3.6.2 Updates on Views

The motivation behind the view mechanism is to tailor how users see the data. Users should not have to worry about the view versus base table distinction. This goal is indeed achieved in the case of queries on views; a view can be used just like any other relation in defining a query. However, it is natural to want to specify updates on views as well. Here, unfortunately, the distinction between a view and a base table must be kept in mind.

The SQL-92 standard allows updates to be specified only on views that are defined on a single base table using just selection and projection, with no use of aggregate operations. Such views are called **updatable views**. This definition is oversimplified, but it captures the spirit of the restrictions. An update on such a restricted view can

always be implemented by updating the underlying base table in an unambiguous way. Consider the following view:

```
CREATE VIEW GoodStudents (sid, gpa)
      AS SELECT S.sid, S.gpa
         FROM   Students S
         WHERE  S.gpa > 3.0
```

We can implement a command to modify the gpa of a GoodStudents row by modifying the corresponding row in Students. We can delete a GoodStudents row by deleting the corresponding row from Students. (In general, if the view did not include a key for the underlying table, several rows in the table could 'correspond' to a single row in the view. This would be the case, for example, if we used *S.sname* instead of *S.sid* in the definition of GoodStudents. A command that affects a row in the view would then affect all corresponding rows in the underlying table.)

We can insert a GoodStudents row by inserting a row into Students, using *null* values in columns of Students that do not appear in GoodStudents (e.g., *sname*, *login*). Note that primary key columns are not allowed to contain *null* values. Therefore, if we attempt to insert rows through a view that does not contain the primary key of the underlying table, the insertions will be rejected. For example, if GoodStudents contained *sname* but not *sid*, we could not insert rows into Students through insertions to GoodStudents.

An important observation is that an `INSERT` or `UPDATE` may change the underlying base table so that the resulting (i.e., inserted or modified) row is not in the view! For example, if we try to insert a row $\langle 51234, 2.8 \rangle$ into the view, this row can be (padded with *null* values in the other fields of Students and then) added to the underlying Students table, but it will not appear in the GoodStudents view because it does not satisfy the view condition $gpa > 3.0$. The SQL-92 default action is to allow this insertion, but we can disallow it by adding the clause `WITH CHECK OPTION` to the definition of the view.

We caution the reader that when a view is defined in terms of another view, the interaction between these view definitions with respect to updates and the `CHECK OPTION` clause can be complex; we will not go into the details.

Need to Restrict View Updates

While the SQL-92 rules on updatable views are more stringent than necessary, there are some fundamental problems with updates specified on views, and there is good reason to limit the class of views that can be updated. Consider the Students relation and a new relation called Clubs:

Clubs(*cname:* `string`, *jyear:* `date`, *mname:* `string`)

A tuple in Clubs denotes that the student called *mname* has been a member of the club *cname* since the date *jyear*.[4] Suppose that we are often interested in finding the names and logins of students with a gpa greater than 3 who belong to at least one club, along with the club name and the date they joined the club. We can define a view for this purpose:

```
CREATE VIEW ActiveStudents (name, login, club, since)
        AS SELECT S.sname, S.login, C.cname, C.jyear
           FROM   Students S, Clubs C
           WHERE  S.sname = C.mname AND S.gpa > 3
```

Consider the instances of Students and Clubs shown in Figures 3.19 and 3.20. When

cname	*jyear*	*mname*
Sailing	1996	Dave
Hiking	1997	Smith
Rowing	1998	Smith

Figure 3.19 An Instance C of Clubs

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8

Figure 3.20 An Instance $S3$ of Students

evaluated using the instances C and $S3$, ActiveStudents contains the rows shown in Figure 3.21.

name	*login*	*club*	*since*
Dave	dave@cs	Sailing	1996
Smith	smith@ee	Hiking	1997
Smith	smith@ee	Rowing	1998
Smith	smith@math	Hiking	1997
Smith	smith@math	Rowing	1998

Figure 3.21 Instance of ActiveStudents

Now suppose that we want to delete the row ⟨*Smith, smith@ee, Hiking, 1997*⟩ from ActiveStudents. How are we to do this? ActiveStudents rows are not stored explicitly but are computed as needed from the Students and Clubs tables using the view definition. So we must change either Students or Clubs (or both) in such a way that evaluating the

[4]We remark that Clubs has a poorly designed schema (chosen for the sake of our discussion of view updates), since it identifies students by name, which is not a candidate key for Students.

view definition on the modified instance does not produce the row ⟨*Smith, smith@ee, Hiking, 1997.*⟩ This task can be accomplished in one of two ways: by either deleting the row ⟨*53688, Smith, smith@ee, 18, 3.2*⟩ from Students or deleting the row ⟨*Hiking, 1997, Smith*⟩ from Clubs. But neither solution is satisfactory. Removing the Students row has the effect of also deleting the row ⟨*Smith, smith@ee, Rowing, 1998*⟩ from the view ActiveStudents. Removing the Clubs row has the effect of also deleting the row ⟨*Smith, smith@math, Hiking, 1997*⟩ from the view ActiveStudents. Neither of these side effects is desirable. In fact, the only reasonable solution is to *disallow* such updates on views.

There are views involving more than one base table that can, in principle, be safely updated. The B-Students view that we introduced at the beginning of this section is an example of such a view. Consider the instance of B-Students shown in Figure 3.18 (with, of course, the corresponding instances of Students and Enrolled as in Figure 3.4). To insert a tuple, say ⟨*Dave, 50000, Reggae203*⟩ B-Students, we can simply insert a tuple ⟨*Reggae203, B, 50000*⟩ into Enrolled since there is already a tuple for *sid* 50000 in Students. To insert ⟨*John, 55000, Reggae203*⟩, on the other hand, we have to insert ⟨*Reggae203, B, 55000*⟩ into Enrolled and also insert ⟨*55000, John,* null, null, null⟩ into Students. Observe how *null* values are used in fields of the inserted tuple whose value is not available. Fortunately, the view schema contains the primary key fields of both underlying base tables; otherwise, we would not be able to support insertions into this view. To delete a tuple from the view B-Students, we can simply delete the corresponding tuple from Enrolled.

Although this example illustrates that the SQL-92 rules on updatable views are unnecessarily restrictive, it also brings out the complexity of handling view updates in the general case. For practical reasons, the SQL-92 standard has chosen to allow only updates on a very restricted class of views.

3.7 DESTROYING/ALTERING TABLES AND VIEWS

If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows *and* remove the table definition information), we can use the `DROP TABLE` command. For example, `DROP TABLE Students RESTRICT` destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword `RESTRICT` is replaced by `CASCADE`, Students is dropped and any referencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be specified. A view can be dropped using the `DROP VIEW` command, which is just like `DROP TABLE`.

`ALTER TABLE` modifies the structure of an existing table. To add a column called *maiden-name* to Students, for example, we would use the following command:

```
ALTER TABLE Students
        ADD COLUMN maiden-name CHAR(10)
```

The definition of Students is modified to add this column, and all existing rows are padded with *null* values in this column. `ALTER TABLE` can also be used to delete columns and to add or drop integrity constraints on a table; we will not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

3.8 POINTS TO REVIEW

- The main element of the relational model is a *relation*. A *relation schema* describes the structure of a relation by specifying the relation name and the names of each field. In addition, the relation schema includes *domain constraints*, which are type restrictions on the fields of the relation. The number of fields is called the *degree* of the relation. The *relation instance* is an actual table that contains a set of *tuples* that adhere to the relation schema. The number of tuples is called the *cardinality* of the relation. SQL-92 is a standard language for interacting with a DBMS. Its *data definition language (DDL)* enables the creation (`CREATE TABLE`) and modification (`DELETE, UPDATE`) of relations. **(Section 3.1)**

- *Integrity constraints* are conditions on a database schema that every legal database instance has to satisfy. Besides domain constraints, other important types of ICs are *key constraints* (a minimal set of fields that uniquely identify a tuple) and *foreign key constraints* (fields in one relation that refer to fields in another relation). SQL-92 supports the specification of the above kinds of ICs, as well as more general constraints called *table constraints* and *assertions*. **(Section 3.2)**

- ICs are enforced whenever a relation is modified and the specified ICs might conflict with the modification. For foreign key constraint violations, SQL-92 provides several alternatives to deal with the violation: `NO ACTION`, `CASCADE`, `SET DEFAULT`, and `SET NULL`. **(Section 3.3)**

- A *relational database query* is a question about the data. SQL supports a very expressive query language. **(Section 3.4)**

- There are standard translations of ER model constructs into SQL. Entity sets are mapped into relations. Relationship sets without constraints are also mapped into relations. When translating relationship sets with constraints, weak entity sets, class hierarchies, and aggregation, the mapping is more complicated. **(Section 3.5)**

- A *view* is a relation whose instance is not explicitly stored but are computed as needed. In addition to enabling logical data independence by defining the external schema through views, views play an important role in restricting access to data for

security reasons. Since views might be defined through complex queries, handling updates specified on views is complicated, and SQL-92 has very stringent rules on when a view is updatable. **(Section 3.6)**

- SQL provides language constructs to modify the structure of tables (`ALTER TABLE`) and to destroy tables and views (`DROP TABLE`). **(Section 3.7)**

EXERCISES

Exercise 3.1 Define the following terms: *relation schema, relational database schema, domain, relation instance, relation cardinality,* and *relation degree.*

Exercise 3.2 How many distinct tuples are in a relation instance with cardinality 22?

Exercise 3.3 Does the relational model, as seen by an SQL query writer, provide physical and logical data independence? Explain.

Exercise 3.4 What is the difference between a candidate key and the primary key for a given relation? What is a superkey?

Exercise 3.5 Consider the instance of the Students relation shown in Figure 3.1.

1. Give an example of an attribute (or set of attributes) that you can deduce is *not* a candidate key, based on this instance being legal.
2. Is there any example of an attribute (or set of attributes) that you can deduce *is* a candidate key, based on this instance being legal?

Exercise 3.6 What is a foreign key constraint? Why are such constraints important? What is referential integrity?

Exercise 3.7 Consider the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In that were defined in Section 1.5.2.

1. List all the foreign key constraints among these relations.
2. Give an example of a (plausible) constraint involving one or more of these relations that is not a primary key or foreign key constraint.

Exercise 3.8 Answer each of the following questions briefly. The questions are based on the following relational schema:

Emp(*eid:* `integer`, *ename:* `string`, *age:* `integer`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`, *pct_time:* `integer`)
Dept(*did:* `integer`, *dname:* `string`, *budget:* `real`, *managerid:* `integer`)

1. Give an example of a foreign key constraint that involves the Dept relation. What are the options for enforcing this constraint when a user attempts to delete a Dept tuple?

2. Write the SQL statements required to create the above relations, including appropriate versions of all primary and foreign key integrity constraints.
3. Define the Dept relation in SQL so that every department is guaranteed to have a manager.
4. Write an SQL statement to add 'John Doe' as an employee with $eid = 101$, $age = 32$ and $salary = 15,000$.
5. Write an SQL statement to give every employee a 10% raise.
6. Write an SQL statement to delete the 'Toy' department. Given the referential integrity constraints you chose for this schema, explain what happens when this statement is executed.

Exercise 3.9 Consider the SQL query whose answer is shown in Figure 3.6.

1. Modify this query so that only the *login* column is included in the answer.
2. If the clause `WHERE` $S.gpa >= 2$ is added to the original query, what is the set of tuples in the answer?

Exercise 3.10 Explain why the addition of `NOT NULL` constraints to the SQL definition of the Manages relation (in Section 3.5.3) would not enforce the constraint that each department must have a manager. What, if anything, is achieved by requiring that the *ssn* field of Manages be non-null?

Exercise 3.11 Suppose that we have a ternary relationship R between entity sets A, B, and C such that A has a key constraint and total participation and B has a key constraint; these are the only constraints. A has attributes $a1$ and $a2$, with $a1$ being the key; B and C are similar. R has no descriptive attributes. Write SQL statements that create tables corresponding to this information so as to capture as many of the constraints as possible. If you cannot capture some constraint, explain why.

Exercise 3.12 Consider the scenario from Exercise 2.2 where you designed an ER diagram for a university database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.13 Consider the university database from Exercise 2.3 and the ER diagram that you designed. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.14 Consider the scenario from Exercise 2.4 where you designed an ER diagram for a company database. Write SQL statements to create the corresponding relations and capture as many of the constraints as possible. If you cannot capture some constraints, explain why.

Exercise 3.15 Consider the Notown database from Exercise 2.5. You have decided to recommend that Notown use a relational database system to store company data. Show the SQL statements for creating relations corresponding to the entity sets and relationship sets in your design. Identify any constraints in the ER diagram that you are unable to capture in the SQL statements and briefly explain why you could not express them.

Exercise 3.16 Translate your ER diagram from Exercise 2.6 into a relational schema, and show the SQL statements needed to create the relations, using only key and null constraints. If your translation cannot capture any constraints in the ER diagram, explain why.

In Exercise 2.6, you also modified the ER diagram to include the constraint that tests on a plane must be conducted by a technician who is an expert on that model. Can you modify the SQL statements defining the relations obtained by mapping the ER diagram to check this constraint?

Exercise 3.17 Consider the ER diagram that you designed for the Prescriptions-R-X chain of pharmacies in Exercise 2.7. Define relations corresponding to the entity sets and relationship sets in your design using SQL.

Exercise 3.18 Write SQL statements to create the corresponding relations to the ER diagram you designed for Exercise 2.8. If your translation cannot capture any constraints in the ER diagram, explain why.

PROJECT-BASED EXERCISES

Exercise 3.19 Create the relations Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets_In in Minibase.

Exercise 3.20 Insert the tuples shown in Figures 3.1 and 3.4 into the relations Students and Enrolled. Create reasonable instances of the other relations.

Exercise 3.21 What integrity constraints are enforced by Minibase?

Exercise 3.22 Run the SQL queries presented in this chapter.

BIBLIOGRAPHIC NOTES

The relational model was proposed in a seminal paper by Codd [156]. Childs [146] and Kuhns [392] foreshadowed some of these developments. Gallaire and Minker's book [254] contains several papers on the use of logic in the context of relational databases. A system based on a variation of the relational model in which the entire database is regarded abstractly as a single relation, called the universal relation, is described in [655]. Extensions of the relational model to incorporate *null* values, which indicate an unknown or missing field value, are discussed by several authors; for example, [280, 335, 542, 662, 691].

Pioneering projects include System R [33, 129] at IBM San Jose Research Laboratory (now IBM Almaden Research Center), Ingres [628] at the University of California at Berkeley, PRTV [646] at the IBM UK Scientific Center in Peterlee, and QBE [702] at IBM T.J. Watson Research Center.

A rich theory underpins the field of relational databases. Texts devoted to theoretical aspects include those by Atzeni and DeAntonellis [38]; Maier [436]; and Abiteboul, Hull, and Vianu [3]. [355] is an excellent survey article.

Integrity constraints in relational databases have been discussed at length. [159] addresses semantic extensions to the relational model, but also discusses integrity, in particular referential integrity. [305] discusses semantic integrity constraints. [168] contains papers that address various aspects of integrity constraints, including in particular a detailed discussion of referential integrity. A vast literature deals with enforcing integrity constraints. [41] compares the cost of enforcing integrity constraints via compile-time, run-time, and post-execution checks. [124] presents an SQL-based language for specifying integrity constraints and identifies conditions under which integrity rules specified in this language can be violated. [624] discusses the technique of integrity constraint checking by query modification. [149] discusses real-time integrity constraints. Other papers on checking integrity constraints in databases include [69, 103, 117, 449]. [593] considers the approach of verifying the correctness of programs that access the database, instead of run-time checks. Note that this list of references is far from complete; in fact, it does not include any of the many papers on checking recursively specified integrity constraints. Some early papers in this widely studied area can be found in [254] and [253].

For references on SQL, see the bibliographic notes for Chapter 5. This book does not discuss specific products based on the relational model, but many fine books do discuss each of the major commercial systems; for example, Chamberlin's book on DB2 [128], Date and McGoveran's book on Sybase [172], and Koch and Loney's book on Oracle [382].

Several papers consider the problem of translating updates specified on views into updates on the underlying table [49, 174, 360, 405, 683]. [250] is a good survey on this topic. See the bibliographic notes for Chapter 23 for references to work querying views and maintaining materialized views.

[642] discusses a design methodology based on developing an ER diagram and then translating to the relational model. Markowitz considers referential integrity in the context of ER to relational mapping and discusses the support provided in some commercial systems (as of that date) in [446, 447].

PART III

DATA STORAGE AND INDEXING

7

STORING DATA: DISKS AND FILES

A memory is what is left when something happens and does not completely unhappen.

—Edward DeBono

This chapter initiates a study of the internals of an RDBMS. In terms of the DBMS architecture presented in Section 1.8, it covers the disk space manager, the buffer manager, and the layer that supports the abstraction of a file of records. Later chapters cover auxiliary structures to speed retrieval of desired subsets of the data, and the implementation of a relational query language.

Data in a DBMS is stored on storage devices such as disks and tapes; we concentrate on disks and cover tapes briefly. The disk space manager is responsible for keeping track of available disk space. The file manager, which provides the abstraction of a file of records to higher levels of DBMS code, issues requests to the disk space manager to obtain and relinquish space on disk. The file management layer requests and frees disk space in units of a **page**; the size of a page is a DBMS parameter, and typical values are 4 KB or 8 KB. The file management layer is responsible for keeping track of the pages in a file and for arranging records within pages.

When a record is needed for processing, it must be fetched from disk to main memory. The page on which the record resides is determined by the file manager. Sometimes, the file manager uses auxiliary data structures to quickly identify the page that contains a desired record. After identifying the required page, the file manager issues a request for the page to a layer of DBMS code called the buffer manager. The buffer manager fetches a requested page from disk into a region of main memory called the buffer pool and tells the file manager the location of the requested page.

We cover the above points in detail in this chapter. Section 7.1 introduces disks and tapes. Section 7.2 describes RAID disk systems. Section 7.3 discusses how a DBMS manages disk space, and Section 7.4 explains how a DBMS fetches data from disk into main memory. Section 7.5 discusses how a collection of pages is organized into a file and how auxiliary data structures can be built to speed up retrieval of records from a file. Section 7.6 covers different ways to arrange a collection of records on a page, and Section 7.7 covers alternative formats for storing individual records.

7.1 THE MEMORY HIERARCHY

Memory in a computer system is arranged in a hierarchy, as shown in Figure 7.1. At the top, we have **primary storage**, which consists of cache and main memory, and provides very fast access to data. Then comes **secondary storage**, which consists of slower devices such as magnetic disks. **Tertiary storage** is the slowest class of storage devices; for example, optical disks and tapes. Currently, the cost of a given amount of

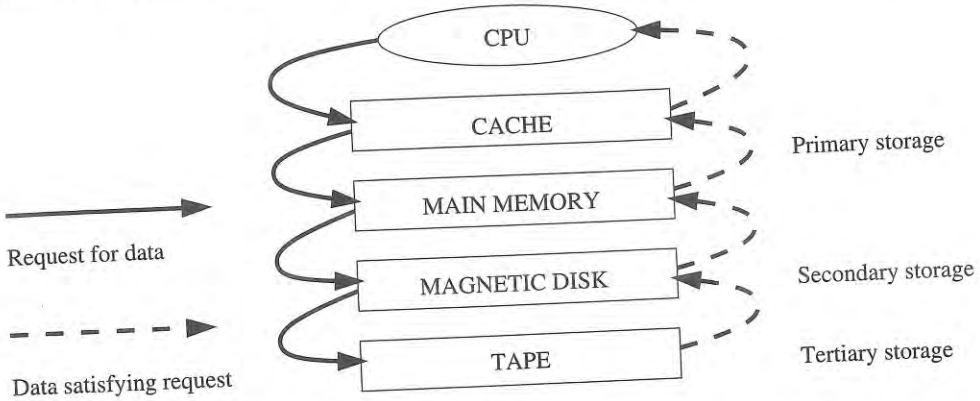


Figure 7.1 The Memory Hierarchy

main memory is about 100 times the cost of the same amount of disk space, and tapes are even less expensive than disks. Slower storage devices such as tapes and disks play an important role in database systems because the amount of data is typically very large. Since buying enough main memory to store all data is prohibitively expensive, we must store data on tapes and disks and build database systems that can retrieve data from lower levels of the memory hierarchy into main memory as needed for processing.

There are reasons other than cost for storing data on secondary and tertiary storage. On systems with 32-bit addressing, only 2^{32} bytes can be directly referenced in main memory; the number of data objects may exceed this number! Further, data must be maintained across program executions. This requires storage devices that retain information when the computer is restarted (after a shutdown or a crash); we call such storage **nonvolatile**. Primary storage is usually volatile (although it is possible to make it nonvolatile by adding a battery backup feature), whereas secondary and tertiary storage is nonvolatile.

Tapes are relatively inexpensive and can store very large amounts of data. They are a good choice for *archival* storage, that is, when we need to maintain data for a long period but do not expect to access it very often. A Quantum DLT 4000 drive is a typical tape device; it stores 20 GB of data and can store about twice as much by compressing the data. It records data on 128 *tape tracks*, which can be thought of as a

linear sequence of adjacent bytes, and supports a sustained transfer rate of 1.5 MB/sec with uncompressed data (typically 3.0 MB/sec with compressed data). A single DLT 4000 tape drive can be used to access up to seven tapes in a stacked configuration, for a maximum compressed data capacity of about 280 GB.

The main drawback of tapes is that they are sequential access devices. We must essentially step through all the data in order and cannot directly access a given location on tape. For example, to access the last byte on a tape, we would have to wind through the entire tape first. This makes tapes unsuitable for storing *operational data*, or data that is frequently accessed. Tapes are mostly used to back up operational data periodically.

7.1.1 Magnetic Disks

Magnetic disks support direct access to a desired location and are widely used for database applications. A DBMS provides seamless access to data on disk; applications need not worry about whether data is in main memory or disk. To understand how disks work, consider Figure 7.2, which shows the structure of a disk in simplified form.

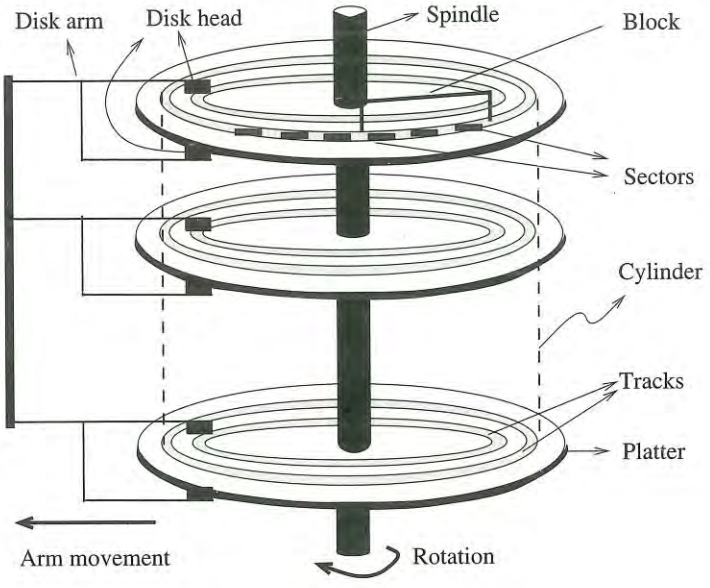


Figure 7.2 Structure of a Disk

Data is stored on disk in units called **disk blocks**. A disk block is a contiguous sequence of bytes and is the unit in which data is written to a disk and read from a disk. Blocks are arranged in concentric rings called **tracks**, on one or more **platters**. Tracks can be recorded on one or both surfaces of a platter; we refer to platters as

single-sided or double-sided accordingly. The set of all tracks with the same diameter is called a **cylinder**, because the space occupied by these tracks is shaped like a cylinder; a cylinder contains one track per platter surface. Each track is divided into arcs called **sectors**, whose size is a characteristic of the disk and cannot be changed. The size of a disk block can be set when the disk is initialized as a multiple of the sector size.

An array of **disk heads**, one per recorded surface, is moved as a unit; when one head is positioned over a block, the other heads are in identical positions with respect to their platters. To read or write a block, a disk head must be positioned on top of the block. As the size of a platter decreases, seek times also decrease since we have to move a disk head a smaller distance. Typical platter diameters are 3.5 inches and 5.25 inches.

Current systems typically allow at most one disk head to read or write at any one time. All the disk heads cannot read or write in parallel—this technique would increase data transfer rates by a factor equal to the number of disk heads, and considerably speed up sequential scans. The reason they cannot is that it is very difficult to ensure that all the heads are perfectly aligned on the corresponding tracks. Current approaches are both expensive and more prone to faults as compared to disks with a single active head. In practice very few commercial products support this capability, and then only in a limited way; for example, two disk heads may be able to operate in parallel.

A **disk controller** interfaces a disk drive to the computer. It implements commands to read or write a sector by moving the arm assembly and transferring data to and from the disk surfaces. A **checksum** is computed for when data is written to a sector and stored with the sector. The checksum is computed again when the data on the sector is read back. If the sector is corrupted or the read is faulty for some reason, it is very unlikely that the checksum computed when the sector is read matches the checksum computed when the sector was written. The controller computes checksums and if it detects an error, it tries to read the sector again. (Of course, it signals a failure if the sector is corrupted and read fails repeatedly.)

While direct access to any desired location in main memory takes approximately the same time, determining the time to access a location on disk is more complicated. The time to access a disk block has several components. **Seek time** is the time taken to move the disk heads to the track on which a desired block is located. **Rotational delay** is the waiting time for the desired block to rotate under the disk head; it is the time required for half a rotation on average and is usually less than seek time. **Transfer time** is the time to actually read or write the data in the block once the head is positioned, that is, the time for the disk to rotate over the block.

An example of a current disk: The IBM Deskstar 14GPX. The IBM Deskstar 14GPX is a 3.5 inch, 14.4 GB hard disk with an average seek time of 9.1 milliseconds (msec) and an average rotational delay of 4.17 msec. However, the time to seek from one tack to the next is just 2.2 msec, the maximum seek time is 15.5 msec. The disk has five double-sided platters that spin at 7,200 rotations per minute. Each platter holds 3.35 GB of data, with a density of 2.6 gigabit per square inch. The data transfer rate is about 13 MB per second. To put these numbers in perspective, observe that a disk access takes about 10 msecs, whereas accessing a main memory location typically takes less than 60 nanoseconds!

7.1.2 Performance Implications of Disk Structure

1. Data must be in memory for the DBMS to operate on it.
2. The unit for data transfer between disk and main memory is a block; if a single item on a block is needed, the entire block is transferred. Reading or writing a disk block is called an **I/O** (for input/output) operation.
3. The time to read or write a block varies, depending on the location of the data:
   access time = seek time + rotational delay + transfer time

These observations imply that the time taken for database operations is affected significantly by how data is stored on disks. The time for moving blocks to or from disk usually dominates the time taken for database operations. To minimize this time, it is necessary to locate data records strategically on disk, because of the geometry and mechanics of disks. In essence, if two records are frequently used together, we should place them close together. The 'closest' that two records can be on a disk is to be on the same block. In decreasing order of closeness, they could be on the same track, the same cylinder, or an adjacent cylinder.

Two records on the same block are obviously as close together as possible, because they are read or written as part of the same block. As the platter spins, other blocks on the track being read or written rotate under the active head. In current disk designs, all the data on a track can be read or written in one revolution. After a track is read or written, another disk head becomes active, and another track in the same cylinder is read or written. This process continues until all tracks in the current cylinder are read or written, and then the arm assembly moves (in or out) to an adjacent cylinder. Thus, we have a natural notion of 'closeness' for blocks, which we can extend to a notion of *next* and *previous* blocks.

Exploiting this notion of next by arranging records so that they are read or written sequentially is very important for reducing the time spent in disk I/Os. Sequential access minimizes seek time and rotational delay and is much faster than random access.

(This observation is reinforced and elaborated in Exercises 7.5 and 7.6, and the reader is urged to work through them.)

7.2 RAID

Disks are potential bottlenecks for system performance and storage system reliability. Even though disk performance has been improving continuously, microprocessor performance has advanced much more rapidly. The performance of microprocessors has improved at about 50 percent or more per year, but disk access times have improved at a rate of about 10 percent per year and disk transfer rates at a rate of about 20 percent per year. In addition, since disks contain mechanical elements, they have much higher failure rates than electronic parts of a computer system. If a disk fails, all the data stored on it is lost.

A **disk array** is an arrangement of several disks, organized so as to increase performance and improve reliability of the resulting storage system. Performance is increased through data striping. Data striping distributes data over several disks to give the impression of having a single large, very fast disk. Reliability is improved through **redundancy**. Instead of having a single copy of the data, redundant information is maintained. The redundant information is carefully organized so that in case of a disk failure, it can be used to reconstruct the contents of the failed disk. Disk arrays that implement a combination of data striping and redundancy are called **redundant arrays of independent disks**, or in short, **RAID**.[1] Several RAID organizations, referred to as **RAID levels**, have been proposed. Each RAID level represents a different trade-off between reliability and performance.

In the remainder of this section, we will first discuss data striping and redundancy and then introduce the RAID levels that have become industry standards.

7.2.1 Data Striping

A disk array gives the user the abstraction of having a single, very large disk. If the user issues an I/O request, we first identify the set of physical disk blocks that store the data requested. These disk blocks may reside on a single disk in the array or may be distributed over several disks in the array. Then the set of blocks is retrieved from the disk(s) involved. Thus, how we distribute the data over the disks in the array influences how many disks are involved when an I/O request is processed.

[1] Historically, the *I* in RAID stood for inexpensive, as a large number of small disks was much more economical than a single very large disk. Today, such very large disks are not even manufactured—a sign of the impact of RAID.

Redundancy schemes: Alternatives to the parity scheme include schemes based on **Hamming codes** and **Reed-Solomon codes**. In addition to recovery from single disk failures, Hamming codes can identify which disk has failed. Reed-Solomon codes can recover from up to two simultaneous disk failures. A detailed discussion of these schemes is beyond the scope of our discussion here; the bibliography provides pointers for the interested reader.

In **data striping**, the data is segmented into equal-size partitions that are distributed over multiple disks. The size of a partition is called the **striping unit**. The partitions are usually distributed using a round robin algorithm: If the disk array consists of D disks, then partition i is written onto disk `i mod k`.

As an example, consider a striping unit of a bit. Since any D successive data bits are spread over all D data disks in the array, all I/O requests involve all disks in the array. Since the smallest unit of transfer from a disk is a block, each I/O request involves transfer of at least D blocks. Since we can read the D blocks from the D disks in parallel, the transfer rate of each request is D times the transfer rate of a single disk; each request uses the aggregated bandwidth of all disks in the array. But the disk access time of the array is basically the access time of a single disk since all disk heads have to move for all requests. Therefore, for a disk array with a striping unit of a single bit, the number of requests per time unit that the array can process and the average response time for each individual request are similar to that of a single disk.

As another example, consider a striping unit of a disk block. In this case, I/O requests of the size of a disk block are processed by one disk in the array. If there are many I/O requests of the size of a disk block and the requested blocks reside on different disks, we can process all requests in parallel and thus reduce the average response time of an I/O request. Since we distributed the striping partitions round-robin, large requests of the size of many contiguous blocks involve all disks. We can process the request by all disks in parallel and thus increase the transfer rate to the aggregated bandwidth of all D disks.

7.2.2 Redundancy

While having more disks increases storage system performance, it also lowers overall storage system reliability. Assume that the **mean-time-to-failure**, or MTTF, of a single disk is 50,000 hours (about 5.7 years). Then, the MTTF of an array of 100 disks is only $50,000/100 = 500$ hours or about 21 days, assuming that failures occur independently and that the failure probability of a disk does not change over time. (Actually, disks have a higher failure probability early and late in their lifetimes. Early failures are often due to undetected manufacturing defects; late failures occur

since the disk wears out. Failures do not occur independently either: consider a fire in the building, an earthquake, or purchase of a set of disks that come from a 'bad' manufacturing batch.)

Reliability of a disk array can be increased by storing redundant information. If a disk failure occurs, the redundant information is used to reconstruct the data on the failed disk. Redundancy can immensely increase the MTTF of a disk array. When incorporating redundancy into a disk array design, we have to make two choices. First, we have to decide where to store the redundant information. We can either store the redundant information on a small number of **check disks** or we can distribute the redundant information uniformly over all disks.

The second choice we have to make is how to compute the redundant information. Most disk arrays store parity information: In the **parity scheme**, an extra check disk contains information that can be used to recover from failure of any one disk in the array. Assume that we have a disk array with D disks and consider the first bit on each data disk. Suppose that i of the D data bits are one. The first bit on the check disk is set to one if i is odd, otherwise it is set to zero. This bit on the check disk is called the **parity** of the data bits. The check disk contains parity information for each set of corresponding D data bits.

To recover the value of the first bit of a failed disk we first count the number of bits that are one on the $D-1$ nonfailed disks; let this number be j. If j is odd and the parity bit is one, or if j is even and the parity bit is zero, then the value of the bit on the failed disk must have been zero. Otherwise, the value of the bit on the failed disk must have been one. Thus, with parity we can recover from failure of any one disk. Reconstruction of the lost information involves reading all data disks and the check disk.

For example, with an additional 10 disks with redundant information, the MTTF of our example storage system with 100 data disks can be increased to more than 250 years! What is more important, a large MTTF implies a small failure probability during the actual usage time of the storage system, which is usually much smaller than the reported lifetime or the MTTF. (Who actually uses 10-year-old disks?)

In a RAID system, the disk array is partitioned into **reliability groups**, where a reliability group consists of a set of *data disks* and a set of *check disks*. A common *redundancy scheme* (see box) is applied to each group. The number of check disks depends on the RAID level chosen. In the remainder of this section, we assume for ease of explanation that there is only one reliability group. The reader should keep in mind that actual RAID implementations consist of several reliability groups, and that the number of groups plays a role in the overall reliability of the resulting storage system.

7.2.3 Levels of Redundancy

Throughout the discussion of the different RAID levels, we consider sample data that would just fit on four disks. That is, without any RAID technology our storage system would consist of exactly four data disks. Depending on the RAID level chosen, the number of additional disks varies from zero to four.

Level 0: Nonredundant

A RAID Level 0 system uses data striping to increase the maximum bandwidth available. No redundant information is maintained. While being the solution with the lowest cost, reliability is a problem, since the MTTF decreases linearly with the number of disk drives in the array. RAID Level 0 has the best write performance of all RAID levels, because absence of redundant information implies that no redundant information needs to be updated! Interestingly, RAID Level 0 does not have the best read performance of all RAID levels, since systems with redundancy have a choice of scheduling disk accesses as explained in the next section.

In our example, the RAID Level 0 solution consists of only four data disks. Independent of the number of data disks, the effective space utilization for a RAID Level 0 system is always 100 percent.

Level 1: Mirrored

A RAID Level 1 system is the most expensive solution. Instead of having one copy of the data, two identical copies of the data on two different disks are maintained. This type of redundancy is often called **mirroring**. Every write of a disk block involves a write on both disks. These writes may not be performed simultaneously, since a global system failure (e.g., due to a power outage) could occur while writing the blocks and then leave both copies in an inconsistent state. Therefore, we always write a block on one disk first and then write the other copy on the mirror disk. Since two copies of each block exist on different disks, we can distribute reads between the two disks and allow *parallel reads* of different disk blocks that conceptually reside on the same disk. A read of a block can be scheduled to the disk that has the smaller expected access time. RAID Level 1 does not stripe the data over different disks, thus the transfer rate for a single request is comparable to the transfer rate of a single disk.

In our example, we need four data and four check disks with mirrored data for a RAID Level 1 implementation. The effective space utilization is 50 percent, independent of the number of data disks.

Level 0+1: Striping and Mirroring

RAID Level 0+1—sometimes also referred to as RAID level 10—combines striping and mirroring. Thus, as in RAID Level 1, read requests of the size of a disk block can be scheduled both to a disk or its mirror image. In addition, read requests of the size of several contiguous blocks benefit from the aggregated bandwidth of all disks. The cost for writes is analogous to RAID Level 1.

As in RAID Level 1, our example with four data disks requires four check disks and the effective space utilization is always 50 percent.

Level 2: Error-Correcting Codes

In RAID Level 2 the striping unit is a single bit. The redundancy scheme used is Hamming code. In our example with four data disks, only three check disks are needed. In general, the number of check disks grows logarithmically with the number of data disks.

Striping at the bit level has the implication that in a disk array with D data disks, the smallest unit of transfer for a read is a set of D blocks. Thus, Level 2 is good for workloads with many large requests since for each request the aggregated bandwidth of all data disks is used. But RAID Level 2 is bad for small requests of the size of an individual block for the same reason. (See the example in Section 7.2.1.) A write of a block involves reading D blocks into main memory, modifying $D + C$ blocks and writing $D + C$ blocks to disk, where C is the number of check disks. This sequence of steps is called a *read-modify-write* cycle.

For a RAID Level 2 implementation with four data disks, three check disks are needed. Thus, in our example the effective space utilization is about 57 percent. The effective space utilization increases with the number of data disks. For example, in a setup with 10 data disks, four check disks are needed and the effective space utilization is 71 percent. In a setup with 25 data disks, five check disks are required and the effective space utilization grows to 83 percent.

Level 3: Bit-Interleaved Parity

While the redundancy schema used in RAID Level 2 improves in terms of cost upon RAID Level 1, it keeps more redundant information than is necessary. Hamming code, as used in RAID Level 2, has the advantage of being able to identify which disk has failed. But disk controllers can easily detect which disk has failed. Thus, the check disks do not need to contain information to identify the failed disk. Information to recover the lost data is sufficient. Instead of using several disks to store Hamming code,

RAID Level 3 has a single check disk with parity information. Thus, the reliability overhead for RAID Level 3 is a single disk, the lowest overhead possible.

The performance characteristics of RAID Level 2 and RAID Level 3 are very similar. RAID Level 3 can also process only one I/O at a time, the minimum transfer unit is D blocks, and a write requires a read-modify-write cycle.

Level 4: Block-Interleaved Parity

RAID Level 4 has a striping unit of a disk block, instead of a single bit as in RAID Level 3. Block-level striping has the advantage that read requests of the size of a disk block can be served entirely by the disk where the requested block resides. Large read requests of several disk blocks can still utilize the aggregated bandwidth of the D disks.

The write of a single block still requires a read-modify-write cycle, but only one data disk and the check disk are involved. The parity on the check disk can be updated without reading all D disk blocks, because the new parity can be obtained by noticing the differences between the old data block and the new data block and then applying the difference to the parity block on the check disk:

$$\texttt{NewParity} = (\texttt{OldData}\ \text{XOR}\ \texttt{NewData})\ \text{XOR}\ \texttt{OldParity}$$

The read-modify-write cycle involves reading of the old data block and the old parity block, modifying the two blocks, and writing them back to disk, resulting in four disk accesses per write. Since the check disk is involved in each write, it can easily become the bottleneck.

RAID Level 3 and 4 configurations with four data disks require just a single check disk. Thus, in our example, the effective space utilization is 80 percent. The effective space utilization increases with the number of data disks, since always only one check disk is necessary.

Level 5: Block-Interleaved Distributed Parity

RAID Level 5 improves upon Level 4 by distributing the parity blocks uniformly over all disks, instead of storing them on a single check disk. This distribution has two advantages. First, several write requests can potentially be processed in parallel, since the bottleneck of a unique check disk has been eliminated. Second, read requests have a higher level of parallelism. Since the data is distributed over all disks, read requests involve all disks, whereas in systems with a dedicated check disk the check disk never participates in reads.

A RAID Level 5 system has the best performance of all RAID levels with redundancy for small and large read and large write requests. Small writes still require a read-modify-write cycle and are thus less efficient than in RAID Level 1.

In our example, the corresponding RAID Level 5 system has 5 disks overall and thus the effective space utilization is the same as in RAID levels 3 and 4.

Level 6: P+Q Redundancy

The motivation for RAID Level 6 is the observation that recovery from failure of a single disk is not sufficient in very large disk arrays. First, in large disk arrays, a second disk might fail before replacement of an already failed disk could take place. In addition, the probability of a disk failure during recovery of a failed disk is not negligible.

A RAID Level 6 system uses Reed-Solomon codes to be able to recover from up to two simultaneous disk failures. RAID Level 6 requires (conceptually) two check disks, but it also uniformly distributes redundant information at the block level as in RAID Level 5. Thus, the performance characteristics for small and large read requests and for large write requests are analogous to RAID Level 5. For small writes, the read-modify-write procedure involves six instead of four disks as compared to RAID Level 5, since two blocks with redundant information need to be updated.

For a RAID Level 6 system with storage capacity equal to four data disks, six disks are required. Thus, in our example, the effective space utilization is 66 percent.

7.2.4 Choice of RAID Levels

If data loss is not an issue, RAID Level 0 improves overall system performance at the lowest cost. RAID Level 0+1 is superior to RAID Level 1. The main application areas for RAID Level 0+1 systems are small storage subsystems where the cost of mirroring is moderate. Sometimes RAID Level 0+1 is used for applications that have a high percentage of writes in their workload, since RAID Level 0+1 provides the best write performance. RAID levels 2 and 4 are always inferior to RAID levels 3 and 5, respectively. RAID Level 3 is appropriate for workloads consisting mainly of large transfer requests of several contiguous blocks. The performance of a RAID Level 3 system is bad for workloads with many small requests of a single disk block. RAID Level 5 is a good general-purpose solution. It provides high performance for large requests as well as for small requests. RAID Level 6 is appropriate if a higher level of reliability is required.

7.3 DISK SPACE MANAGEMENT

The lowest level of software in the DBMS architecture discussed in Section 1.8, called the **disk space manager**, manages space on disk. Abstractly, the disk space manager supports the concept of a **page** as a unit of data, and provides commands to allocate or deallocate a page and read or write a page. The size of a page is chosen to be the size of a disk block and pages are stored as disk blocks so that reading or writing a page can be done in one disk I/O.

It is often useful to allocate a sequence of pages as a *contiguous* sequence of blocks to hold data that is frequently accessed in sequential order. This capability is essential for exploiting the advantages of sequentially accessing disk blocks, which we discussed earlier in this chapter. Such a capability, if desired, must be provided by the disk space manager to higher-level layers of the DBMS.

Thus, the disk space manager hides details of the underlying hardware (and possibly the operating system) and allows higher levels of the software to think of the data as a collection of pages.

7.3.1 Keeping Track of Free Blocks

A database grows and shrinks as records are inserted and deleted over time. The disk space manager keeps track of which disk blocks are in use, in addition to keeping track of which pages are on which disk blocks. Although it is likely that blocks are initially allocated sequentially on disk, subsequent allocations and deallocations could in general create 'holes.'

One way to keep track of block usage is to maintain a list of free blocks. As blocks are deallocated (by the higher-level software that requests and uses these blocks), we can add them to the free list for future use. A pointer to the first block on the free block list is stored in a known location on disk.

A second way is to maintain a bitmap with one bit for each disk block, which indicates whether a block is in use or not. A bitmap also allows very fast identification and allocation of contiguous areas on disk. This is difficult to accomplish with a linked list approach.

7.3.2 Using OS File Systems to Manage Disk Space

Operating systems also manage space on disk. Typically, an operating system supports the abstraction of a *file as a sequence of bytes*. The OS manages space on the disk and translates requests such as "Read byte i of file f" into corresponding low-level

instructions: "Read block m of track t of cylinder c of disk d." A database disk space manager could be built using OS files. For example, the entire database could reside in one or more OS files for which a number of blocks are allocated (by the OS) and initialized. The disk space manager is then responsible for managing the space in these OS files.

Many database systems do not rely on the OS file system and instead do their own disk management, either from scratch or by extending OS facilities. The reasons are practical as well as technical. One practical reason is that a DBMS vendor who wishes to support several OS platforms cannot assume features specific to any OS, for portability, and would therefore try to make the DBMS code as self-contained as possible. A technical reason is that on a 32-bit system, the largest file size is 4 GB, whereas a DBMS may want to access a single file larger than that. A related problem is that typical OS files cannot span disk devices, which is often desirable or even necessary in a DBMS. Additional technical reasons why a DBMS does not rely on the OS file system are outlined in Section 7.4.2.

7.4 BUFFER MANAGER

To understand the role of the buffer manager, consider a simple example. Suppose that the database contains 1,000,000 pages, but only 1,000 pages of main memory are available for holding data. Consider a query that requires a scan of the entire file. Because all the data cannot be brought into main memory at one time, the DBMS must bring pages into main memory as they are needed and, in the process, decide what existing page in main memory to replace to make space for the new page. The policy used to decide which page to replace is called the **replacement policy**.

In terms of the DBMS architecture presented in Section 1.8, the **buffer manager** is the software layer that is responsible for bringing pages from disk to main memory as needed. The buffer manager manages the available main memory by partitioning it into a collection of pages, which we collectively refer to as the **buffer pool**. The main memory pages in the buffer pool are called **frames**; it is convenient to think of them as slots that can hold a page (that usually resides on disk or other secondary storage media).

Higher levels of the DBMS code can be written without worrying about whether data pages are in memory or not; they ask the buffer manager for the page, and it is brought into a frame in the buffer pool if it is not already there. Of course, the higher-level code that requests a page must also release the page when it is no longer needed, by informing the buffer manager, so that the frame containing the page can be reused. The higher-level code must also inform the buffer manager if it modifies the requested page; the buffer manager then makes sure that the change is propagated to the copy of the page on disk. Buffer management is illustrated in Figure 7.3.

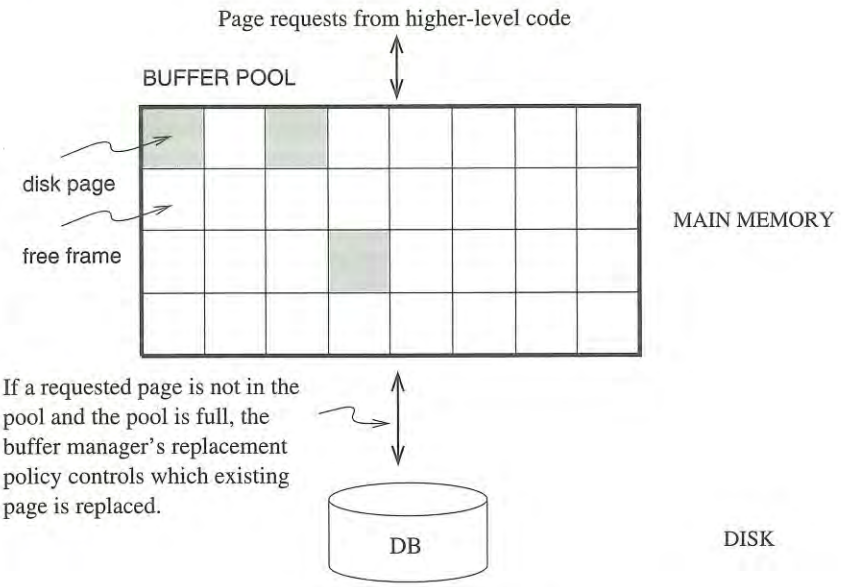


Figure 7.3 The Buffer Pool

In addition to the buffer pool itself, the buffer manager maintains some bookkeeping information, and two variables for each frame in the pool: *pin_count* and *dirty*. The number of times that the page currently in a given frame has been requested but not released—the number of current users of the page—is recorded in the *pin_count* variable for that frame. The boolean variable *dirty* indicates whether the page has been modified since it was brought into the buffer pool from disk.

Initially, the *pin_count* for every frame is set to 0, and the *dirty* bits are turned off. When a page is requested the buffer manager does the following:

1. Checks the buffer pool to see if it contains the requested page. If the page is not in the pool, the buffer manager brings it in as follows:
   (a) Chooses a frame for replacement, using the replacement policy, and increments its *pin_count*.
   (b) If the *dirty* bit for the replacement frame is on, writes the page it contains to disk (that is, the disk copy of the page is overwritten with the contents of the frame).
   (c) Reads the requested page into the replacement frame.
2. Returns the (main memory) address of the replacement frame to the requestor.

Incrementing *pin_count* is often called **pinning** the requested page in its frame. When the code that calls the buffer manager and requests the page subsequently calls the

buffer manager and releases the page, the *pin_count* of the frame containing the requested page is decremented. This is called **unpinning** the page. If the requestor has modified the page, it also informs the buffer manager of this at the time that it unpins the page, and the *dirty* bit for the frame is set. The buffer manager will not read another page into a frame until its *pin_count* becomes 0, that is, until all requestors of the page have unpinned it.

If a requested page is not in the buffer pool, and if a free frame is not available in the buffer pool, a frame with *pin_count* 0 is chosen for replacement. If there are many such frames, a frame is chosen according to the buffer manager's replacement policy. We discuss various replacement policies in Section 7.4.1.

When a page is eventually chosen for replacement, if the *dirty* bit is not set, it means that the page has not been modified since being brought into main memory. Thus, there is no need to write the page back to disk; the copy on disk is identical to the copy in the frame, and the frame can simply be overwritten by the newly requested page. Otherwise, the modifications to the page must be propagated to the copy on disk. (The crash recovery protocol may impose further restrictions, as we saw in Section 1.7. For example, in the Write-Ahead Log (WAL) protocol, special log records are used to describe the changes made to a page. The log records pertaining to the page that is to be replaced may well be in the buffer; if so, the protocol requires that they be written to disk *before* the page is written to disk.)

If there is no page in the buffer pool with *pin_count* 0 and a page that is not in the pool is requested, the buffer manager must wait until some page is released before responding to the page request. In practice, the transaction requesting the page may simply be aborted in this situation! So pages should be released—by the code that calls the buffer manager to request the page—as soon as possible.

A good question to ask at this point is "What if a page is requested by several different transactions?" That is, what if the page is requested by programs executing independently on behalf of different users? There is the potential for such programs to make conflicting changes to the page. The locking protocol (enforced by higher-level DBMS code, in particular the transaction manager) ensures that each transaction obtains a shared or exclusive lock before requesting a page to read or modify. Two different transactions cannot hold an exclusive lock on the same page at the same time; this is how conflicting changes are prevented. The buffer manager simply assumes that the appropriate lock has been obtained before a page is requested.

7.4.1 Buffer Replacement Policies

The policy that is used to choose an unpinned page for replacement can affect the time taken for database operations considerably. Many alternative policies exist, and each is suitable in different situations.

The best-known replacement policy is **least recently used** (LRU). This can be implemented in the buffer manager using a queue of pointers to frames with *pin_count* 0. A frame is added to the end of the queue when it becomes a candidate for replacement (that is, when the *pin_count* goes to 0). The page chosen for replacement is the one in the frame at the head of the queue.

A variant of LRU, called **clock** replacement, has similar behavior but less overhead. The idea is to choose a page for replacement using a *current* variable that takes on values 1 through N, where N is the number of buffer frames, in circular order. We can think of the frames being arranged in a circle, like a clock's face, and *current* as a clock hand moving across the face. In order to approximate LRU behavior, each frame also has an associated *referenced* bit, which is turned on when the page *pin_count* goes to 0.

The *current* frame is considered for replacement. If the frame is not chosen for replacement, *current* is incremented and the next frame is considered; this process is repeated until some frame is chosen. If the *current* frame has *pin_count* greater than 0, then it is not a candidate for replacement and *current* is incremented. If the *current* frame has the *referenced* bit turned on, the clock algorithm turns the *referenced* bit off and increments *current*—this way, a recently referenced page is less likely to be replaced. If the *current* frame has *pin_count* 0 and its *referenced* bit is off, then the page in it is chosen for replacement. If all frames are pinned in some sweep of the clock hand (that is, the value of *current* is incremented until it repeats), this means that no page in the buffer pool is a replacement candidate.

The LRU and clock policies are not always the best replacement strategies for a database system, particularly if many user requests require sequential scans of the data. Consider the following illustrative situation. Suppose the buffer pool has 10 frames, and the file to be scanned has 10 or fewer pages. Assuming, for simplicity, that there are no competing requests for pages, only the first scan of the file does any I/O. Page requests in subsequent scans will always find the desired page in the buffer pool. On the other hand, suppose that the file to be scanned has 11 pages (which is one more than the number of available pages in the buffer pool). Using LRU, every scan of the file will result in reading every page of the file! In this situation, called **sequential flooding**, LRU is the *worst* possible replacement strategy.

Buffer management in practice: IBM DB2 and Sybase ASE allow buffers to be partitioned into named pools. Each database, table, or index can be bound to one of these pools. Each pool can be configured to use either LRU or clock replacement in ASE; DB2 uses a variant of clock replacement, with the initial clock value based on the nature of the page (e.g., index nonleaves get a higher starting clock value, which delays their replacement). Interestingly, a buffer pool client in DB2 can explicitly indicate that it *hates* a page, making the page the next choice for replacement. As a special case, DB2 applies MRU for the pages fetched in some utility operations (e.g., RUNSTATS), and DB2 V6 also supports FIFO. Informix and Oracle 7 both maintain a single global buffer pool using LRU; Microsoft SQL Server has a single pool using clock replacement. In Oracle 8, tables can be bound to one of two pools; one has high priority, and the system attempts to keep pages in this pool in memory.

Beyond setting a maximum number of pins for a given transaction, there are typically no features for controlling buffer pool usage on a per-transaction basis. Microsoft SQL Server, however, supports a reservation of buffer pages by queries that require large amounts of memory (e.g., queries involving sorting or hashing).

Other replacement policies include **first in first out** (FIFO) and **most recently used** (MRU), which also entail overhead similar to LRU, and **random**, among others. The details of these policies should be evident from their names and the preceding discussion of LRU and clock.

7.4.2 Buffer Management in DBMS versus OS

Obvious similarities exist between virtual memory in operating systems and buffer management in database management systems. In both cases the goal is to provide access to more data than will fit in main memory, and the basic idea is to bring in pages from disk to main memory as needed, replacing pages that are no longer needed in main memory. Why can't we build a DBMS using the virtual memory capability of an OS? A DBMS can often predict the order in which pages will be accessed, or **page reference patterns**, much more accurately than is typical in an OS environment, and it is desirable to utilize this property. Further, a DBMS needs more control over when a page is written to disk than an OS typically provides.

A DBMS can often predict reference patterns because most page references are generated by higher-level operations (such as sequential scans or particular implementations of various relational algebra operators) with a known pattern of page accesses. This ability to predict reference patterns allows for a better choice of pages to replace and makes the idea of specialized buffer replacement policies more attractive in the DBMS environment.

Prefetching: In IBM DB2, both sequential and list prefetch (prefetching a list of pages) are supported. In general, the prefetch size is 32 4KB pages, but this can be set by the user. For some sequential type database utilities (e.g., COPY, RUNSTATS), DB2 will prefetch upto 64 4KB pages. For a smaller buffer pool (i.e., less than 1000 buffers), the prefetch quantity is adjusted downward to 16 or 8 pages. Prefetch size can be configured by the user; for certain environments, it may be best to prefetch 1000 pages at a time! Sybase ASE supports asynchronous prefetching of upto 256 pages, and uses this capability to reduce latency during indexed access to a table in a range scan. Oracle 8 uses prefetching for sequential scan, retrieving large objects, and for certain index scans. Microsoft SQL Server supports prefetching for sequential scan and for scans along the leaf level of a B+ tree index and the prefetch size can be adjusted as a scan progresses. SQL Server also uses asynchronous prefetching extensively. Informix supports prefetching with a user-defined prefetch size.

Even more important, being able to predict reference patterns enables the use of a simple and very effective strategy called **prefetching of pages**. The buffer manager can anticipate the next several page requests and fetch the corresponding pages into memory *before* the pages are requested. This strategy has two benefits. First, the pages are available in the buffer pool when they are requested. Second, reading in a contiguous block of pages is much faster than reading the same pages at different times in response to distinct requests. (Review the discussion of disk geometry to appreciate why this is so.) If the pages to be prefetched are not contiguous, recognizing that several pages need to be fetched can nonetheless lead to faster I/O because an order of retrieval can be chosen for these pages that minimizes seek times and rotational delays.

Incidentally, note that the I/O can typically be done concurrently with CPU computation. Once the prefetch request is issued to the disk, the disk is responsible for reading the requested pages into memory pages and the CPU can continue to do other work.

A DBMS also requires the ability to explicitly *force* a page to disk, that is, to ensure that the copy of the page on disk is updated with the copy in memory. As a related point, a DBMS must be able to ensure that certain pages in the buffer pool are written to disk *before* certain other pages are written, in order to implement the WAL protocol for crash recovery, as we saw in Section 1.7. Virtual memory implementations in operating systems cannot be relied upon to provide such control over when pages are written to disk; the OS command to write a page to disk may be implemented by essentially recording the write request, and deferring the actual modification of the disk copy. If the system crashes in the interim, the effects can be catastrophic for a DBMS. (Crash recovery is discussed further in Chapter 20.)

7.5 FILES AND INDEXES

We now turn our attention from the way pages are stored on disk and brought into main memory to the way pages are used to store records and organized into logical collections or **files**. Higher levels of the DBMS code treat a page as effectively being a collection of records, ignoring the representation and storage details. In fact, the concept of a collection of records is not limited to the contents of a single page; a **file of records** is a collection of records that may reside on several pages. In this section, we consider how a collection of pages can be organized as a file. We discuss how the space on a page can be organized to store a collection of records in Sections 7.6 and 7.7.

Each record has a unique identifier called a **record id**, or **rid** for short. As we will see in Section 7.6, we can identify the page containing a record by using the record's rid. The basic file structure that we consider, called a *heap file*, stores records in random order and supports retrieval of all records or retrieval of a particular record specified by its rid. Sometimes we want to retrieve records by specifying some condition on the fields of desired records, for example, "Find all employee records with *age* 35." To speed up such selections, we can build auxiliary data structures that allow us to quickly find the rids of employee records that satisfy the given selection condition. Such an auxiliary structure is called an index; we introduce indexes in Section 7.5.2.

7.5.1 Heap Files

The simplest file structure is an unordered file or **heap file**. The data in the pages of a heap file is not ordered in any way, and the only guarantee is that one can retrieve all records in the file by repeated requests for the next record. Every record in the file has a unique rid, and every page in a file is of the same size.

Supported operations on a heap file include *create* and *destroy* files, *insert* a record, *delete* a record with a given rid, *get* a record with a given rid, and *scan* all records in the file. To get or delete a record with a given rid, note that we must be able to find the id of the page containing the record, given the id of the record.

We must keep track of the pages in each heap file in order to support scans, and we must keep track of pages that contain free space in order to implement insertion efficiently. We discuss two alternative ways to maintain this information. In each of these alternatives, pages must hold two pointers (which are page ids) for file-level bookkeeping in addition to the data.

Linked List of Pages

One possibility is to maintain a heap file as a doubly linked list of pages. The DBMS can remember where the first page is located by maintaining a table containing pairs of ⟨*heap_file_name*, *page_1_addr*⟩ in a known location on disk. We call the first page of the file the *header page*.

An important task is to maintain information about empty slots created by deleting a record from the heap file. This task has two distinct parts: how to keep track of free space within a page and how to keep track of pages that have some free space. We consider the first part in Section 7.6. The second part can be addressed by maintaining a doubly linked list of pages with free space and a doubly linked list of full pages; together, these lists contain *all* pages in the heap file. This organization is illustrated in Figure 7.4; note that each pointer is really a page id.

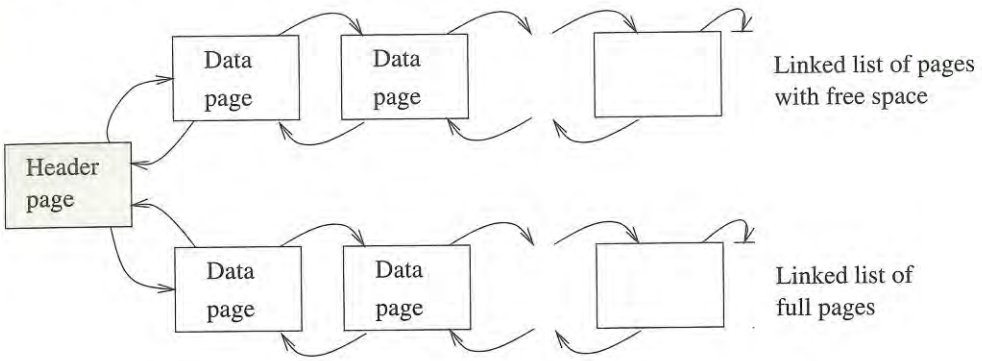


Figure 7.4 Heap File Organization with a Linked List

If a new page is required, it is obtained by making a request to the disk space manager and then added to the list of pages in the file (probably as a page with free space, because it is unlikely that the new record will take up all the space on the page). If a page is to be deleted from the heap file, it is removed from the list and the disk space manager is told to deallocate it. (Note that the scheme can easily be generalized to allocate or deallocate a sequence of several pages and maintain a doubly linked list of these page sequences.)

One disadvantage of this scheme is that virtually all pages in a file will be on the free list if records are of variable length, because it is likely that every page has at least a few free bytes. To insert a typical record, we must retrieve and examine several pages on the free list before we find one with enough free space. The directory-based heap file organization that we discuss next addresses this problem.

Directory of Pages

An alternative to a linked list of pages is to maintain a **directory of pages**. The DBMS must remember where the first directory page of each heap file is located. The directory is itself a collection of pages and is shown as a linked list in Figure 7.5. (Other organizations are possible for the directory itself, of course.)

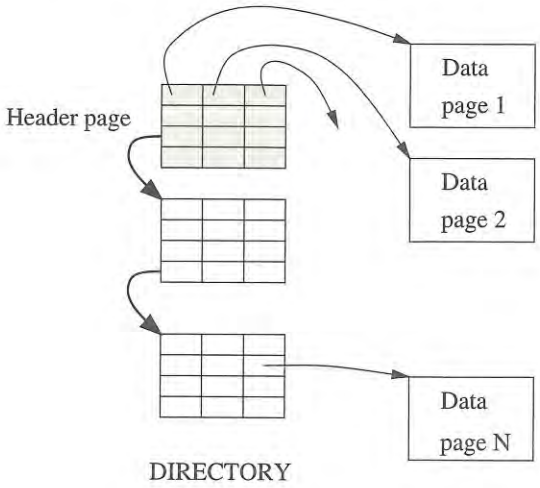


Figure 7.5 Heap File Organization with a Directory

Each directory entry identifies a page (or a sequence of pages) in the heap file. As the heap file grows or shrinks, the number of entries in the directory—and possibly the number of pages in the directory itself—grows or shrinks correspondingly. Note that since each directory entry is quite small in comparison to a typical page, the size of the directory is likely to be very small in comparison to the size of the heap file.

Free space can be managed by maintaining a bit per entry, indicating whether the corresponding page has any free space, or a count per entry, indicating the amount of free space on the page. If the file contains variable-length records, we can examine the free space count for an entry to determine if the record will fit on the page pointed to by the entry. Since several entries fit on a directory page, we can efficiently search for a data page with enough space to hold a record that is to be inserted.

7.5.2 Introduction to Indexes

Sometimes we want to find all records that have a given value in a particular field. If we can find the rids of all such records, we can locate the page containing each record from the record's rid; however, the heap file organization does not help us to find the

rids of such records. An **index** is an auxiliary data structure that is intended to help us find rids of records that meet a selection condition.

Consider how you locate a desired book in a library. You can search a collection of index cards, sorted on author name or book title, to find the call number for the book. Because books are stored according to call numbers, the call number enables you to walk to the shelf that contains the book you need. Observe that an index on author name cannot be used to locate a book by title, and vice versa; each index speeds up certain kinds of searches, but not all. This is illustrated in Figure 7.6.

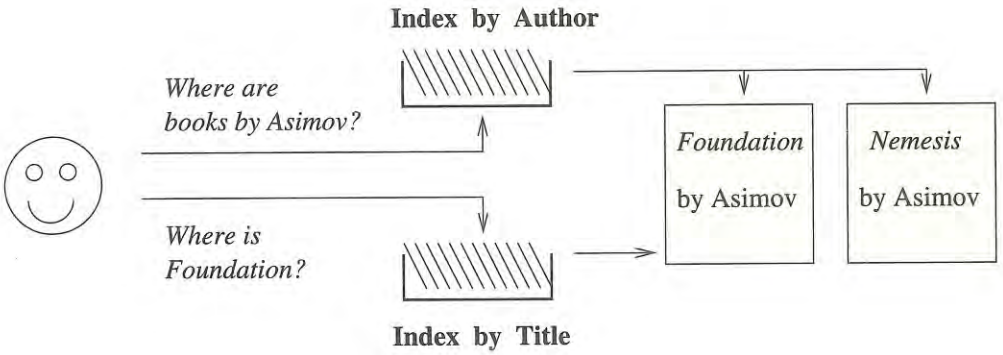


Figure 7.6 Indexes in a Library

The same ideas apply when we want to support efficient retrieval of a desired subset of the data in a file. From an implementation standpoint, an index is just another kind of file, containing records that direct traffic on requests for data records. Every index has an associated **search key**, which is a collection of one or more fields of the file of records for which we are building the index; any subset of the fields can be a search key. We sometimes refer to the file of records as the **indexed file**.

An index is designed to speed up equality or range selections on the search key. For example, if we wanted to build an index to improve the efficiency of queries about employees of a given age, we could build an index on the *age* attribute of the employee dataset. The records stored in an index file, which we refer to as **entries** to avoid confusion with data records, allow us to find data records with a given search key value. In our example the index might contain $\langle age, rid \rangle$ pairs, where *rid* identifies a data record.

The pages in the index file are organized in some way that allows us to quickly locate those entries in the index that have a given search key value. For example, we have to find entries with $age \geq 30$ (and then follow the rids in the retrieved entries) in order to find employee records for employees who are older than 30. Organization techniques, or data structures, for index files are called **access methods**, and several are known,

Rids in commercial systems: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all implement record ids as a page id and slot number. Sybase ASE uses the following page organization, which is typical: Pages contain a header followed by the rows and a slot array. The header contains the page identity, its allocation state, page free space state, and a timestamp. The slot array is simply a mapping of slot number to page offset.
Oracle 8 and SQL Server use logical record ids rather than page id and slot number in one special case: If a table has a clustered index, then records in the table are identified using the key value for the clustered index. This has the advantage that secondary indexes don't have to be reorganized if records are moved across pages.

including B+ trees (Chapter 9) and hash-based structures (Chapter 10). B+ tree index files and hash-based index files are built using the page allocation and manipulation facilities provided by the disk space manager, just like heap files.

7.6 PAGE FORMATS *

The page abstraction is appropriate when dealing with I/O issues, but higher levels of the DBMS see data as a collection of records. In this section, we consider how a collection of records can be arranged on a page. We can think of a page as a collection of **slots**, each of which contains a record. A record is identified by using the pair ⟨*page id*, *slot number*⟩; this is the record id (rid). (We remark that an alternative way to identify records is to assign each record a unique integer as its rid and to maintain a table that lists the page and slot of the corresponding record for each rid. Due to the overhead of maintaining this table, the approach of using ⟨*page id*, *slot number*⟩ as an rid is more common.)

We now consider some alternative approaches to managing slots on a page. The main considerations are how these approaches support operations such as searching, inserting, or deleting records on a page.

7.6.1 Fixed-Length Records

If all records on the page are guaranteed to be of the same length, record slots are uniform and can be arranged consecutively within a page. At any instant, some slots are occupied by records, and others are unoccupied. When a record is inserted into the page, we must locate an empty slot and place the record there. The main issues are how we keep track of empty slots and how we locate all records on a page. The alternatives hinge on how we handle the deletion of a record.

The first alternative is to store records in the first N slots (where N is the number of records on the page); whenever a record is deleted, we move the last record on the page into the vacated slot. This format allows us to locate the ith record on a page by a simple offset calculation, and all empty slots appear together at the end of the page. However, this approach does not work if there are external references to the record that is moved (because the rid contains the slot number, which is now changed).

The second alternative is to handle deletions by using an array of bits, one per slot, to keep track of free slot information. Locating records on the page requires scanning the bit array to find slots whose bit is on; when a record is deleted, its bit is turned off. The two alternatives for storing fixed-length records are illustrated in Figure 7.7. Note that in addition to the information about records on the page, a page usually contains additional file-level information (e.g., the id of the next page in the file). The figure does not show this additional information.

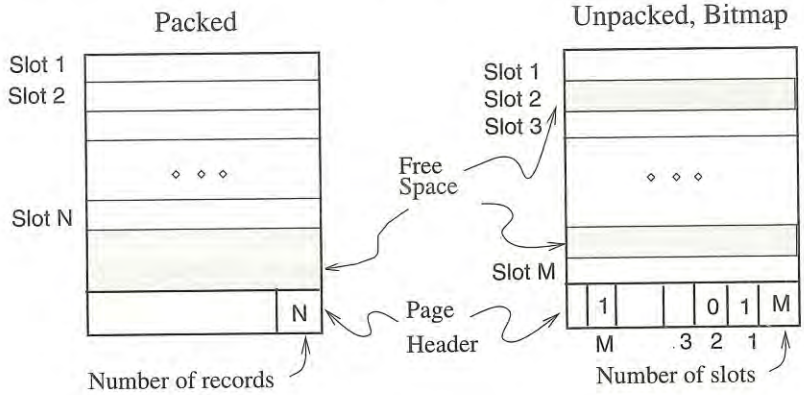


Figure 7.7 Alternative Page Organizations for Fixed-Length Records

The *slotted page* organization described for variable-length records in Section 7.6.2 can also be used for fixed-length records. It becomes attractive if we need to move records around on a page for reasons other than keeping track of space freed by deletions. A typical example is that we want to keep the records on a page sorted (according to the value in some field).

7.6.2 Variable-Length Records

If records are of variable length, then we cannot divide the page into a fixed collection of slots. The problem is that when a new record is to be inserted, we have to find an empty slot of just the right length—if we use a slot that is too big, we waste space, and obviously we cannot use a slot that is smaller than the record length. Therefore, when a record is inserted, we must allocate just the right amount of space for it, and when a record is deleted, we must move records to fill the hole created by the deletion,

in order to ensure that all the free space on the page is contiguous. Thus, the ability to move records on a page becomes very important.

The most flexible organization for variable-length records is to maintain a **directory of slots** for each page, with a ⟨*record offset, record length*⟩ pair per slot. The first component (*record offset*) is a 'pointer' to the record, as shown in Figure 7.8; it is the offset in bytes from the start of the data area on the page to the start of the record. Deletion is readily accomplished by setting the record offset to -1. Records can be moved around on the page because the rid, which is the page number and slot number (that is, position in the directory), does not change when the record is moved; only the record offset stored in the slot changes.

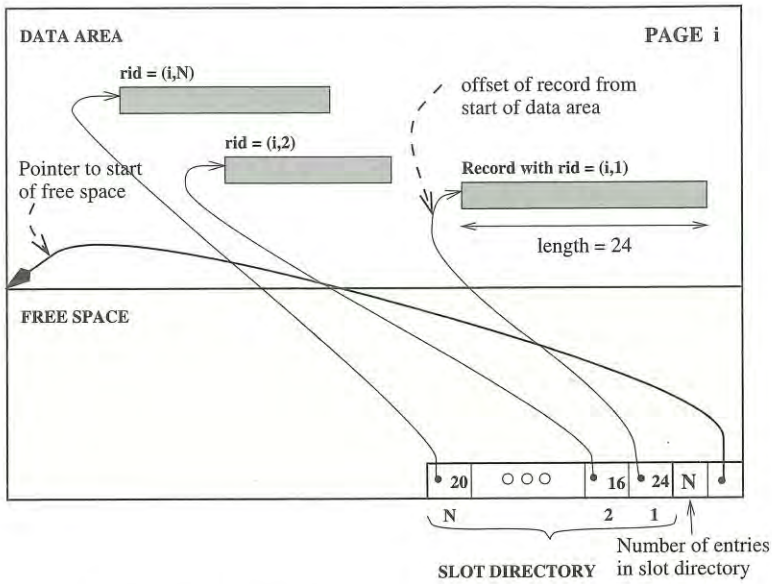


Figure 7.8 Page Organization for Variable-Length Records

The space available for new records must be managed carefully because the page is not preformatted into slots. One way to manage free space is to maintain a pointer (that is, offset from the start of the data area on the page) that indicates the start of the free space area. When a new record is too large to fit into the remaining free space, we have to move records on the page to reclaim the space freed by records that have been deleted earlier. The idea is to ensure that after reorganization, all records appear contiguously, followed by the available free space.

A subtle point to be noted is that the slot for a deleted record cannot always be removed from the slot directory, because slot numbers are used to identify records—by deleting a slot, we change (decrement) the slot number of subsequent slots in the slot directory, and thereby change the rid of records pointed to by subsequent slots. The

only way to remove slots from the slot directory is to remove the last slot if the record that it points to is deleted. However, when a record is inserted, the slot directory should be scanned for an element that currently does not point to any record, and this slot should be used for the new record. A new slot is added to the slot directory only if all existing slots point to records. If inserts are much more common than deletes (as is typically the case), the number of entries in the slot directory is likely to be very close to the actual number of records on the page.

This organization is also useful for fixed-length records if we need to move them around frequently; for example, when we want to maintain them in some sorted order. Indeed, when all records are the same length, instead of storing this common length information in the slot for each record, we can store it once in the system catalog.

In some special situations (e.g., the internal pages of a B+ tree, which we discuss in Chapter 9), we may not care about changing the rid of a record. In this case the slot directory can be compacted after every record deletion; this strategy guarantees that the number of entries in the slot directory is the same as the number of records on the page. If we do not care about modifying rids, we can also sort records on a page in an efficient manner by simply moving slot entries rather than actual records, which are likely to be much larger than slot entries.

A simple variation on the slotted organization is to maintain only record offsets in the slots. For variable-length records, the length is then stored with the record (say, in the first bytes). This variation makes the slot directory structure for pages with fixed-length records be the same as for pages with variable-length records.

7.7 RECORD FORMATS *

In this section we discuss how to organize fields within a record. While choosing a way to organize the fields of a record, we must take into account whether the fields of the record are of fixed or variable length and consider the cost of various operations on the record, including retrieval and modification of fields.

Before discussing record formats, we note that in addition to storing individual records, information that is common to all records of a given record type (such as the number of fields and field types) is stored in the **system catalog**, which can be thought of as a description of the contents of a database, maintained by the DBMS (Section 13.2). This avoids repeated storage of the same information with each record of a given type.

Record formats in commercial systems: In IBM DB2, fixed length fields are at fixed offsets from the beginning of the record. Variable length fields have offset and length in the fixed offset part of the record, and the fields themselves follow the fixed length part of the record. Informix, Microsoft SQL Server, and Sybase ASE use the same organization with minor variations. In Oracle 8, records are structured as if all fields are potentially variable length; a record is a sequence of length–data pairs, with a special length value used to denote a *null* value.

7.7.1 Fixed-Length Records

In a fixed-length record, each field has a fixed length (that is, the value in this field is of the same length in all records), and the number of fields is also fixed. The fields of such a record can be stored consecutively, and, given the address of the record, the address of a particular field can be calculated using information about the lengths of preceding fields, which is available in the system catalog. This record organization is illustrated in Figure 7.9.

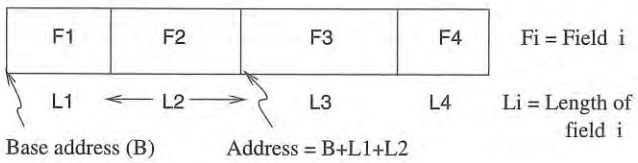


Figure 7.9 Organization of Records with Fixed-Length Fields

7.7.2 Variable-Length Records

In the relational model, every record in a relation contains the same number of fields. If the number of fields is fixed, a record is of variable length only because some of its fields are of variable length.

One possible organization is to store fields consecutively, separated by delimiters (which are special characters that do not appear in the data itself). This organization requires a scan of the record in order to locate a desired field.

An alternative is to reserve some space at the beginning of a record for use as an array of integer offsets—the ith integer in this array is the starting address of the ith field value relative to the start of the record. Note that we also store an offset to the end of the record; this offset is needed to recognize where the last field ends. Both alternatives are illustrated in Figure 7.10.

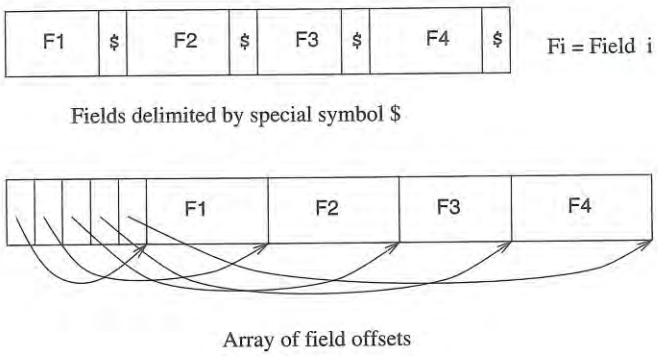


Figure 7.10 Alternative Record Organizations for Variable-Length Fields

The second approach is typically superior. For the overhead of the offset array, we get direct access to any field. We also get a clean way to deal with **null** values. A *null* value is a special value used to denote that the value for a field is unavailable or inapplicable. If a field contains a *null* value, the pointer to the end of the field is set to be the same as the pointer to the beginning of the field. That is, no space is used for representing the *null* value, and a comparison of the pointers to the beginning and the end of the field is used to determine that the value in the field is *null*.

Variable-length record formats can obviously be used to store fixed-length records as well; sometimes, the extra overhead is justified by the added flexibility, because issues such as supporting *null* values and adding fields to a record type arise with fixed-length records as well.

Having variable-length fields in a record can raise some subtle issues, especially when a record is modified.

- Modifying a field may cause it to grow, which requires us to shift all subsequent fields to make space for the modification in all three record formats presented above.
- A record that is modified may no longer fit into the space remaining on its page. If so, it may have to be moved to another page. If rids, which are used to 'point' to a record, include the page number (see Section 7.6), moving a record to another page causes a problem. We may have to leave a 'forwarding address' on this page identifying the new location of the record. And to ensure that space is always available for this forwarding address, we would have to allocate some minimum space for each record, regardless of its length.
- A record may grow so large that it no longer fits on *any* one page. We have to deal with this condition by breaking a record into smaller records. The smaller

Large records in real systems: In Sybase ASE, a record can be at most 1962 bytes. This limit is set by the 2 KB log page size, since records are not allowed to be larger than a page. The exceptions to this rule are BLOBs and CLOBs, which consist of a set of bidirectionally linked pages. IBM DB2 and Microsoft SQL Server also do not allow records to span pages, although large objects are allowed to span pages and are handled separately from other data types. In DB2, record size is limited only by the page size; in SQL Server, a record can be at most 8 KB, excluding LOBs. Informix and Oracle 8 allow records to span pages. Informix allows records to be at most 32 KB, while Oracle has no maximum record size; large records are organized as a singly directed list.

records could be chained together—part of each smaller record is a pointer to the next record in the chain—to enable retrieval of the entire original record.

7.8 POINTS TO REVIEW

- Memory in a computer system is arranged into *primary storage* (cache and main memory), *secondary storage* (magnetic disks), and *tertiary storage* (optical disks and tapes). Storage devices that store data persistently are called *nonvolatile*. **(Section 7.1)**
- Disks provide inexpensive, nonvolatile storage. The unit of transfer from disk into main memory is called a *block* or *page*. Blocks are arranged on *tracks* on several *platters*. The time to access a page depends on its location on disk. The access time has three components: the time to move the disk arm to the desired track (*seek time*), the time to wait for the desired block to rotate under the disk head (*rotational delay*), and the time to transfer the data (*transfer time*). **(Section 7.1.1)**
- Careful placement of pages on the disk to exploit the geometry of a disk can minimize the seek time and rotational delay when pages are read sequentially. **(Section 7.1.2)**
- A *disk array* is an arrangement of several disks that are attached to a computer. Performance of a disk array can be increased through *data striping* and reliability can be increased through *redundancy*. Different RAID organizations called *RAID levels* represent different trade-offs between reliability and performance. **(Section 7.2)**
- In a DBMS, the *disk space manager* manages space on disk by keeping track of free and used disk blocks. It also provides the abstraction of the data being a collection of disk pages. DBMSs rarely use OS files for performance, functionality, and portability reasons. **(Section 7.3)**

- In a DBMS, all page requests are centrally processed by the *buffer manager*. The buffer manager transfers pages between the disk and a special area of main memory called the *buffer pool*, which is divided into page-sized chunks called *frames*. For each page in the buffer pool, the buffer manager maintains a *pin_count*, which indicates the number of users of the current page, and a *dirty* flag, which indicates whether the page has been modified. A requested page is kept in the buffer pool until it is released (unpinned) by all users. Subsequently, a page is written back to disk (if it has been modified while in the buffer pool) when the frame containing it is chosen for replacement. **(Section 7.4)**

- The choice of frame to replace is based on the buffer manager's *replacement policy*, for example LRU or clock. Repeated scans of a file can cause *sequential flooding* if LRU is used. **(Section 7.4.1)**

- A DBMS buffer manager can often predict the access pattern for disk pages. It takes advantage of such opportunities by issuing requests to the disk to *prefetch* several pages at a time. This technique minimizes disk arm movement and reduces I/O time. A DBMS also needs to be able to *force* a page to disk to ensure crash recovery. **(Section 7.4.2)**

- Database pages are organized into files, and higher-level DBMS code views the data as a *collection of records*. **(Section 7.5)**

- The simplest file structure is a *heap file*, which is an unordered collection of records. Heap files are either organized as a linked list of data pages or as a list of directory pages that refer to the actual pages with data. **(Section 7.5.1)**

- *Indexes* are auxiliary structures that support efficient retrieval of records based on the values of a *search key*. **(Section 7.5.2)**

- A page contains a collection of *slots*, each of which identifies a record. Slotted pages allow a record to be moved around on a page without altering the record identifier or *rid*, a ⟨*page id*, *slot number*⟩ pair. Efficient page organizations exist for either fixed-length records (bitmap of free slots) or variable-length records (slot directory). **(Section 7.6)**

- For fixed-length records, the fields can be stored consecutively and the address of a field can be easily calculated. Variable-length records can be stored with an array of offsets at the beginning of the record or the individual can be fields separated by a delimiter symbol. The organization with an array of offsets offers direct access to fields (which can be important if records are long and contain many fields) and support for *null* values. **(Section 7.7)**

EXERCISES

Exercise 7.1 What is the most important difference between a disk and a tape?

Exercise 7.2 Explain the terms *seek time, rotational delay,* and *transfer time.*

Exercise 7.3 Both disks and main memory support direct access to any desired location (page). On average, main memory accesses are faster, of course. What is the other important difference (from the perspective of the time required to access a desired page)?

Exercise 7.4 If you have a large file that is frequently scanned sequentially, explain how you would store the pages in the file on a disk.

Exercise 7.5 Consider a disk with a sector size of 512 bytes, 2,000 tracks per surface, 50 sectors per track, 5 double-sided platters, average seek time of 10 msec.

1. What is the capacity of a track in bytes? What is the capacity of each surface? What is the capacity of the disk?
2. How many cylinders does the disk have?
3. Give examples of valid block sizes. Is 256 bytes a valid block size? 2,048? 51,200?
4. If the disk platters rotate at 5,400 rpm (revolutions per minute), what is the maximum rotational delay?
5. Assuming that one track of data can be transferred per revolution, what is the transfer rate?

Exercise 7.6 Consider again the disk specifications from Exercise 7.5 and suppose that a block size of 1,024 bytes is chosen. Suppose that a file containing 100,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

1. How many records fit onto a block?
2. How many blocks are required to store the entire file? If the file is arranged sequentially on disk, how many surfaces are needed?
3. How many records of 100 bytes each can be stored using this disk?
4. If pages are stored sequentially on disk, with page 1 on block 1 of track 1, what is the page stored on block 1 of track 1 on the next disk surface? How would your answer change if the disk were capable of reading/writing from all heads in parallel?
5. What is the time required to read a file containing 100,000 records of 100 bytes each sequentially? Again, how would your answer change if the disk were capable of reading/writing from all heads in parallel (and the data was arranged optimally)?
6. What is the time required to read a file containing 100,000 records of 100 bytes each in some random order? Note that in order to read a record, the block containing the record has to be fetched from disk. Assume that each block request incurs the average seek time and rotational delay.

Exercise 7.7 Explain what the buffer manager must do to process a read request for a page. What happens if the requested page is in the pool but not pinned?

Exercise 7.8 When does a buffer manager write a page to disk?

Exercise 7.9 What does it mean to say that a page is *pinned* in the buffer pool? Who is responsible for pinning pages? Who is responsible for unpinning pages?

Exercise 7.10 When a page in the buffer pool is modified, how does the DBMS ensure that this change is propagated to disk? (Explain the role of the buffer manager as well as the modifier of the page.)

Exercise 7.11 What happens if there is a page request when all pages in the buffer pool are dirty?

Exercise 7.12 What is *sequential flooding* of the buffer pool?

Exercise 7.13 Name an important capability of a DBMS buffer manager that is not supported by a typical operating system's buffer manager.

Exercise 7.14 Explain the term *prefetching*. Why is it important?

Exercise 7.15 Modern disks often have their own main memory caches, typically about one MB, and use this to do prefetching of pages. The rationale for this technique is the empirical observation that if a disk page is requested by some (not necessarily database!) application, 80 percent of the time the next page is requested as well. So the disk gambles by reading ahead.

1. Give a nontechnical reason that a DBMS may not want to rely on prefetching controlled by the disk.
2. Explain the impact on the disk's cache of several queries running concurrently, each scanning a different file.
3. Can the above problem be addressed by the DBMS buffer manager doing its own prefetching? Explain.
4. Modern disks support *segmented caches*, with about four to six segments, each of which is used to cache pages from a different file. Does this technique help, with respect to the above problem? Given this technique, does it matter whether the DBMS buffer manager also does prefetching?

Exercise 7.16 Describe two possible record formats. What are the trade-offs between them?

Exercise 7.17 Describe two possible page formats. What are the trade-offs between them?

Exercise 7.18 Consider the page format for variable-length records that uses a slot directory.

1. One approach to managing the slot directory is to use a maximum size (i.e., a maximum number of slots) and to allocate the directory array when the page is created. Discuss the pros and cons of this approach with respect to the approach discussed in the text.
2. Suggest a modification to this page format that would allow us to sort records (according to the value in some field) without moving records and without changing the record ids.

Exercise 7.19 Consider the two internal organizations for heap files (using lists of pages and a directory of pages) discussed in the text.

1. Describe them briefly and explain the trade-offs. Which organization would you choose if records are variable in length?
2. Can you suggest a single page format to implement both internal file organizations?

Exercise 7.20 Consider a list-based organization of the pages in a heap file in which two lists are maintained: a list of *all* pages in the file and a list of all pages with free space. In contrast, the list-based organization discussed in the text maintains a list of full pages and a list of pages with free space.

1. What are the trade-offs, if any? Is one of them clearly superior?
2. For each of these organizations, describe a page format that can be used to implement it.

Exercise 7.21 Modern disk drives store more sectors on the outer tracks than the inner tracks. Since the rotation speed is constant, the sequential data transfer rate is also higher on the outer tracks. The seek time and rotational delay are unchanged. Considering this information, explain good strategies for placing files with the following kinds of access patterns:

1. Frequent, random accesses to a small file (e.g., catalog relations).
2. Sequential scans of a large file (e.g., selection from a relation with no index).
3. Random accesses to a large file via an index (e.g., selection from a relation via the index).
4. Sequential scans of a small file.

PROJECT-BASED EXERCISES

Exercise 7.22 Study the public interfaces for the disk space manager, the buffer manager, and the heap file layer in Minibase.

1. Are heap files with variable-length records supported?
2. What page format is used in Minibase heap files?
3. What happens if you insert a record whose length is greater than the page size?
4. How is free space handled in Minibase?
5. *Note to Instructors: See Appendix B for additional project-based exercises.*

BIBLIOGRAPHIC NOTES

Salzberg [564] and Wiederhold [681] discuss secondary storage devices and file organizations in detail.

RAID was originally proposed by Patterson, Gibson, and Katz [512]. The article by Chen et al. provides an excellent survey of RAID [144] . Books about RAID include Gibson's dissertation [269] and the publications from the RAID Advisory Board [527].

The design and implementation of storage managers is discussed in [54, 113, 413, 629, 184]. With the exception of [184], these systems emphasize *extensibility*, and the papers contain much of interest from that standpoint as well. Other papers that cover storage management issues in the context of significant implemented prototype systems are [415] and [513]. The Dali storage manager, which is optimized for main memory databases, is described in [345]. Three techniques for implementing long fields are compared in [83].

Stonebraker discusses operating systems issues in the context of databases in [626]. Several buffer management policies for database systems are compared in [150]. Buffer management is also studied in [101, 142, 223, 198].

PART V

DATABASE DESIGN

15

SCHEMA REFINEMENT AND NORMAL FORMS

It is a melancholy truth that even great men have their poor relations.

—Charles Dickens

Conceptual database design gives us a set of relation schemas and integrity constraints (ICs) that can be regarded as a good starting point for the final database design. This initial design must be refined by taking the ICs into account more fully than is possible with just the ER model constructs and also by considering performance criteria and typical workloads. In this chapter we discuss how ICs can be used to refine the conceptual schema produced by translating an ER model design into a collection of relations. Workload and performance considerations are discussed in Chapter 16.

We concentrate on an important class of constraints called *functional dependencies*. Other kinds of ICs, for example *multivalued dependencies* and *join dependencies*, also provide useful information. They can sometimes reveal redundancies that cannot be detected using functional dependencies alone. We discuss these other constraints briefly.

This chapter is organized as follows. Section 15.1 is an overview of the schema refinement approach discussed in this chapter. We introduce functional dependencies in Section 15.2. In Section 15.3 we present several examples that highlight the problems caused by redundancy and illustrate how relational schemas obtained by translating an ER model design can nonetheless suffer from these problems. Thus, ER design is a good starting point, but we still need techniques to detect schemas with these problems and to refine such schemas to eliminate the problems. We lay the foundation for developing such schema refinement techniques in Section 15.4, where we show how to reason with functional dependency information to infer additional dependencies from a given set of dependencies.

We introduce normal forms for relations in Section 15.5; the normal form satisfied by a relation is a measure of the redundancy in the relation. A relation with redundancy can be refined by *decomposing it*, or replacing it with smaller relations that contain the same information, but without redundancy. We discuss decompositions and desirable properties of decompositions in Section 15.6. We show how relations can be decomposed into smaller relations that are in desirable normal forms in Section 15.7. Finally, we discuss the use of other kinds of dependencies for database design in Section 15.8.

15.1 INTRODUCTION TO SCHEMA REFINEMENT

We now present an overview of the problems that schema refinement is intended to address and a refinement approach based on decompositions. Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

15.1.1 Problems Caused by Redundancy

Storing the same information **redundantly**, that is, in more than one place within a database, can lead to several problems:

- **Redundant storage:** Some information is stored repeatedly.
- **Update anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.
- **Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.
- **Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

Consider a relation obtained by translating a variant of the Hourly_Emps entity set from Chapter 2:

Hourly_Emps(*ssn*, *name*, *lot*, *rating*, *hourly_wages*, *hours_worked*)

In this chapter we will omit attribute type information for brevity, since our focus is on the grouping of attributes into relations. We will often abbreviate an attribute name to a single letter and refer to a relation schema by a string of letters, one per attribute. For example, we will refer to the Hourly_Emps schema as *SNLRWH* (*W* denotes the *hourly_wages* attribute).

The key for Hourly_Emps is *ssn*. In addition, suppose that the *hourly_wages* attribute is determined by the *rating* attribute. That is, for a given *rating* value, there is only one permissible *hourly_wages* value. This IC is an example of a *functional dependency*. It leads to possible redundancy in the relation Hourly_Emps, as illustrated in Figure 15.1.

If the same value appears in the *rating* column of two tuples, the IC tells us that the same value must appear in the *hourly_wages* column as well. This redundancy has several negative consequences:

ssn	*name*	*lot*	*rating*	*hourly_wages*	*hours_worked*
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Figure 15.1 An Instance of the Hourly_Emps Relation

- Some information is stored multiple times. For example, the rating value 8 corresponds to the hourly wage 10, and this association is repeated three times. In addition to wasting space by storing the same information many times, redundancy leads to potential inconsistency. For example, the *hourly_wages* in the first tuple could be updated without making a similar change in the second tuple, which is an example of an *update anomaly*. Also, we cannot insert a tuple for an employee unless we know the hourly wage for the employee's rating value, which is an example of an *insertion anomaly*.

- If we delete all tuples with a given rating value (e.g., we delete the tuples for Smethurst and Guldu) we lose the association between that *rating* value and its *hourly_wage* value (a *deletion anomaly*).

Let us consider whether the use of *null* values can address some of these problems. Clearly, *null* values cannot help eliminate redundant storage or update anomalies. It appears that they can address insertion and deletion anomalies. For instance, to deal with the insertion anomaly example, we can insert an employee tuple with *null* values in the hourly wage field. However, *null* values cannot address all insertion anomalies. For example, we cannot record the hourly wage for a rating unless there is an employee with that rating, because we cannot store a null value in the *ssn* field, which is a primary key field. Similarly, to deal with the deletion anomaly example, we might consider storing a tuple with *null* values in all fields except *rating* and *hourly_wages* if the last tuple with a given *rating* would otherwise be deleted. However, this solution will not work because it requires the *ssn* value to be *null*, and primary key fields cannot be *null*. Thus, *null* values do not provide a general solution to the problems of redundancy, even though they can help in some cases. We will not discuss the use of *null* values further.

Ideally, we want schemas that do not permit redundancy, but at the very least we want to be able to identify schemas that do allow redundancy. Even if we choose to accept a schema with some of these drawbacks, perhaps owing to performance considerations, we want to make an informed decision.

15.1.2 Use of Decompositions

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies (and, for that matter, other ICs) can be used to identify such situations and to suggest refinements to the schema. The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of 'smaller' relations. Each of the smaller relations contains a (strict) subset of the attributes of the original relation. We refer to this process as *decomposition* of the larger relation into the smaller relations.

We can deal with the redundancy in Hourly_Emps by decomposing it into two relations:

Hourly_Emps2(*ssn*, *name*, *lot*, *rating*, *hours_worked*)
Wages(*rating*, *hourly_wages*)

The instances of these relations corresponding to the instance of Hourly_Emps relation in Figure 15.1 is shown in Figure 15.2.

ssn	*name*	*lot*	*rating*	*hours_worked*
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

rating	*hourly_wages*
8	10
5	7

Figure 15.2 Instances of Hourly_Emps2 and Wages

Note that we can easily record the hourly wage for any rating simply by adding a tuple to Wages, even if no employee with that rating appears in the current instance of Hourly_Emps. Changing the wage associated with a rating involves updating a single Wages tuple. This is more efficient than updating several tuples (as in the original design), and it also eliminates the potential for inconsistency. Notice that the insertion and deletion anomalies have also been eliminated.

15.1.3 Problems Related to Decomposition

Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several *normal forms* have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation).

With respect to the second question, two properties of decompositions are of particular interest. The *lossless-join* property enables us to recover any instance of the decomposed relation from corresponding instances of the smaller relations. The *dependency-preservation* property enables us to enforce any constraint on the original relation by simply enforcing some contraints on each of the smaller relations. That is, we need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.

A serious drawback of decompositions is that queries over the original relation may require us to join the decomposed relations. If such queries are common, the performance penalty of decomposing the relation may not be acceptable. In this case we may choose to live with some of the problems of redundancy and not decompose the relation. It is important to be aware of the potential problems caused by such residual redundancy in the design and to take steps to avoid them (e.g., by adding some checks to application code). We will not discuss the impact of decompositions on query performance in this chapter; this issue is covered in Section 16.8.

Our goal in this chapter is to explain some powerful concepts and design guidelines based on the theory of functional dependencies. A good database designer should have a firm grasp of normal forms and what problems they (do or do not) alleviate, the technique of decomposition, and potential problems with decompositions. For example, a designer will often ask questions such as these: Is a relation in a given normal form? Is a decomposition dependency-preserving? Our objective is to explain when to raise these questions and the significance of the answers.

15.2 FUNCTIONAL DEPENDENCIES

A **functional dependency** (FD) is a kind of IC that generalizes the concept of a *key*. Let R be a relation schema and let X and Y be nonempty sets of attributes in R. We say that an instance r of R satisfies the FD $X \rightarrow Y$ [1] if the following holds for every pair of tuples t_1 and t_2 in r:

If $t1.X = t2.X$, then $t1.Y = t2.Y$.

We use the notation $t1.X$ to refer to the projection of tuple t_1 onto the attributes in X, in a natural extension of our TRC notation (see Chapter 4) $t.a$ for referring to attribute a of tuple t. An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attributes X, they must also agree on the values in attributes Y.

Figure 15.3 illustrates the meaning of the FD $AB \rightarrow C$ by showing an instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint: Although the FD is not violated, AB is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the A field or the B field, they can differ in the C field without violating the FD. On the other hand, if we add a tuple $\langle a1, b1, c2, d1 \rangle$ to the instance shown in this figure, the resulting instance would violate the FD; to see this violation, compare the first tuple in the figure with the new tuple.

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

Figure 15.3 An Instance that Satisfies $AB \rightarrow C$

Recall that a *legal* instance of a relation must satisfy all specified ICs, including all specified FDs. As noted in Section 3.2, ICs must be identified and specified based on the semantics of the real-world enterprise being modeled. By looking at an instance of a relation, we might be able to tell that a certain FD does *not* hold. However, we can never deduce that an FD *does* hold by looking at one or more instances of the relation because an FD, like other ICs, is a statement about *all* possible legal instances of the relation.

[1] $X \rightarrow Y$ is read as X *functionally determines* Y, or simply as X *determines* Y.

A primary key constraint is a special case of an FD. The attributes in the key play the role of X, and the set of all attributes in the relation plays the role of Y. Note, however, that the definition of an FD does not require that the set X be minimal; the additional minimality condition must be met for X to be a key. If $X \rightarrow Y$ holds, where Y is the set of all attributes, and there is some subset V of X such that $V \rightarrow Y$ holds, then X is a *superkey*; if V is a strict subset of X, then X is not a key.

In the rest of this chapter, we will see several examples of FDs that are not key constraints.

15.3 EXAMPLES MOTIVATING SCHEMA REFINEMENT

It is natural to ask whether we need to decompose relations produced by translating an ER diagram. Shouldn't a good ER design lead to a collection of good relations? Unfortunately, ER design can generate some schemas with redundancy problems, because it is a complex, subjective process, and certain constraints are not expressible in terms of ER diagrams. The examples in this section are intended to illustrate why decomposition of relations produced through ER design might be necessary.

15.3.1 Constraints on an Entity Set

Consider the Hourly_Emps relation again. The constraint that attribute *ssn* is a key can be expressed as an FD:

$$\{ssn\} \rightarrow \{ssn,\ name,\ lot,\ rating,\ hourly\_wages,\ hours\_worked\}$$

For brevity, we will write this FD as $S \rightarrow SNLRWH$, using a single letter to denote each attribute and omitting the set braces, but the reader should remember that both sides of an FD contain sets of attributes. In addition, the constraint that the *hourly_wages* attribute is determined by the *rating* attribute is an FD: $R \rightarrow W$.

As we saw in Section 15.1.1, this FD led to redundant storage of rating–wage associations. *It cannot be expressed in terms of the ER model. Only FDs that determine all attributes of a relation (i.e., key constraints) can be expressed in the ER model.* Therefore, we could not detect it when we considered Hourly_Emps as an entity set during ER modeling.

We could argue that the problem with the original design was an artifact of a poor ER design, which could have been avoided by introducing an entity set called Wage_Table (with attributes *rating* and *hourly_wages*) and a relationship set Has_Wages associating Hourly_Emps and Wage_Table. The point, however, is that we could easily arrive at the original design given the subjective nature of ER modeling. Having formal techniques to identify the problem with this design, and to guide us to a better design, is very

useful. The value of such techniques cannot be underestimated when designing large schemas—schemas with more than a hundred tables are not uncommon!

15.3.2 Constraints on a Relationship Set

The previous example illustrated how FDs can help to refine the subjective decisions made during ER design, but one could argue that the best possible ER diagram would have led to the same final set of relations. Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

We revisit an example from Chapter 2. Suppose that we have entity sets Parts, Suppliers, and Departments, as well as a relationship set Contracts that involves all of them. We refer to the schema for Contracts as *CQPSD*. A contract with contract id *C* specifies that a supplier *S* will supply some quantity *Q* of a part *P* to a department *D*. (We have added the contract id field *C* to the version of the Contracts relation that was discussed in Chapter 2.)

We might have a policy that a department purchases at most one part from any given supplier. Thus, if there are several contracts between the same supplier and department, we know that the same part must be involved in all of them. This constraint is an FD, $DS \rightarrow P$.

Again we have redundancy and its associated problems. We can address this situation by decomposing Contracts into two relations with attributes *CQSD* and *SDP*. Intuitively, the relation *SDP* records the part supplied to a department by a supplier, and the relation *CQSD* records additional information about a contract. It is unlikely that we would arrive at such a design solely through ER modeling, since it is hard to formulate an entity or relationship that corresponds naturally to *CQSD*.

15.3.3 Identifying Attributes of Entities

This example illustrates how a careful examination of FDs can lead to a better understanding of the entities and relationships underlying the relational tables; in particular, it shows that attributes can easily be associated with the 'wrong' entity set during ER design. The ER diagram in Figure 15.4 shows a relationship set called Works_In that is similar to the Works_In relationship set of Chapter 2, but with an additional key constraint indicating that an employee can work in at most one department. (Observe the arrow connecting Employees to Works_In.)

Using the key constraint, we can translate this ER diagram into two relations:

Workers(*ssn*, *name*, *lot*, *did*, *since*)

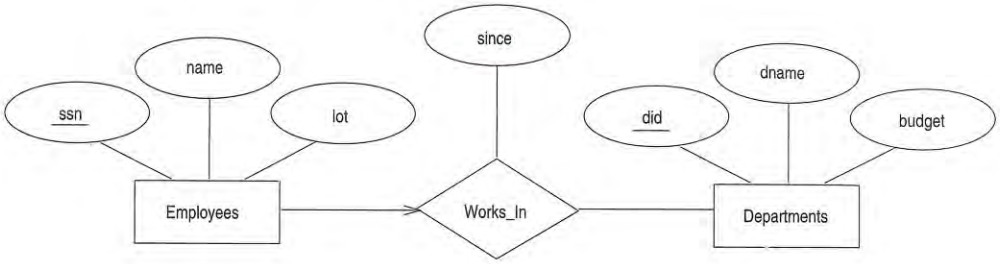


Figure 15.4 The Works_In Relationship Set

Departments(*did*, *dname*, *budget*)

The entity set Employees and the relationship set Works_In are mapped to a single relation, Workers. This translation is based on the second approach discussed in Section 2.4.1.

Now suppose that employees are assigned parking lots based on their department, and that all employees in a given department are assigned the same lot. This constraint is not expressible with respect to the ER diagram of Figure 15.4. It is another example of an FD: $did \rightarrow lot$. The redundancy in this design can be eliminated by decomposing the Workers relation into two relations:

Workers2(*ssn*, *name*, *did*, *since*)
Dept_Lots(*did*, *lot*)

The new design has much to recommend it. We can change the lots associated with a department by updating a single tuple in the second relation (i.e., no update anomalies). We can associate a lot with a department even if it currently has no employees, without using *null* values (i.e., no deletion anomalies). We can add an employee to a department by inserting a tuple to the first relation even if there is no lot associated with the employee's department (i.e., no insertion anomalies).

Examining the two relations Departments and Dept_Lots, which have the same key, we realize that a Departments tuple and a Dept_Lots tuple with the same key value describe the same entity. This observation is reflected in the ER diagram shown in Figure 15.5.

Translating this diagram into the relational model would yield:

Workers2(*ssn*, *name*, *did*, *since*)
Departments(*did*, *dname*, *budget*, *lot*)

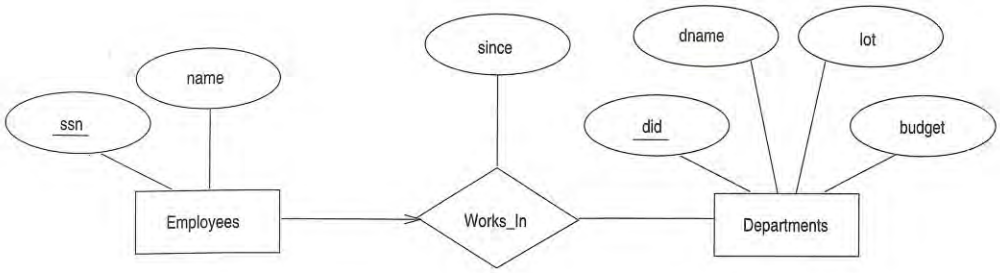


Figure 15.5 Refined Works_In Relationship Set

It seems intuitive to associate lots with employees; on the other hand, the ICs reveal that in this example lots are really associated with departments. The subjective process of ER modeling could miss this point. The rigorous process of normalization would not.

15.3.4 Identifying Entity Sets

Consider a variant of the Reserves schema used in earlier chapters. Let Reserves contain attributes S, B, and D as before, indicating that sailor S has a reservation for boat B on day D. In addition, let there be an attribute C denoting the credit card to which the reservation is charged. We use this example to illustrate how FD information can be used to refine an ER design. In particular, we discuss how FD information can help to decide whether a concept should be modeled as an entity or as an attribute.

Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD $S \rightarrow C$. This constraint indicates that in relation Reserves, we store the credit card number for a sailor as often as we have reservations for that sailor, and we have redundancy and potential update anomalies. A solution is to decompose Reserves into two relations with attributes SBD and SC. Intuitively, one holds information about reservations, and the other holds information about credit cards.

It is instructive to think about an ER design that would lead to these relations. One approach is to introduce an entity set called Credit_Cards, with the sole attribute *cardno*, and a relationship set Has_Card associating Sailors and Credit_Cards. By noting that each credit card belongs to a single sailor, we can map Has_Card and Credit_Cards to a single relation with attributes SC. We would probably not model credit card numbers as entities if our main interest in card numbers is to indicate how a reservation is to be paid for; it suffices to use an attribute to model card numbers in this situation.

A second approach is to make *cardno* an attribute of Sailors. But this approach is not very natural—a sailor may have several cards, and we are not interested in all of them. Our interest is in the one card that is used to pay for reservations, which is best modeled as an attribute of the relationship Reserves.

A helpful way to think about the design problem in this example is that we first make *cardno* an attribute of Reserves and then refine the resulting tables by taking into account the FD information. (Whether we refine the design by adding *cardno* to the table obtained from Sailors or by creating a new table with attributes SC is a separate issue.)

15.4 REASONING ABOUT FUNCTIONAL DEPENDENCIES

The discussion up to this point has highlighted the need for techniques that allow us to carefully examine and further refine relations obtained through ER design (or, for that matter, through other approaches to conceptual design). Before proceeding with the main task at hand, which is the discussion of such schema refinement techniques, we digress to examine FDs in more detail because they play such a central role in schema analysis and refinement.

Given a set of FDs over a relation schema R, there are typically several additional FDs that hold over R whenever all of the given FDs hold. As an example, consider:

Workers(*ssn*, *name*, *lot*, *did*, *since*)

We know that *ssn* $\rightarrow$ *did* holds, since *ssn* is the key, and FD *did* $\rightarrow$ *lot* is given to hold. Therefore, in any legal instance of Workers, if two tuples have the same *ssn* value, they must have the same *did* value (from the first FD), and because they have the same *did* value, they must also have the same *lot* value (from the second FD). Thus, the FD *ssn* $\rightarrow$ *lot* also holds on Workers.

We say that an FD f **is implied by** a given set F of FDs if f holds on every relation instance that satisfies all dependencies in F, that is, f holds whenever all FDs in F hold. Note that it is not sufficient for f to hold on some instance that satisfies all dependencies in F; rather, f must hold on *every* instance that satisfies all dependencies in F.

15.4.1 Closure of a Set of FDs

The set of all FDs implied by a given set F of FDs is called the **closure of F** and is denoted as F^+. An important question is how we can **infer**, or compute, the closure of a given set F of FDs. The answer is simple and elegant. The following three rules, called **Armstrong's Axioms**, can be applied repeatedly to infer all FDs implied by

a set F of FDs. We use X, Y, and Z to denote *sets* of attributes over a relation schema R:

- **Reflexivity:** If $X \supseteq Y$, then $X \rightarrow Y$.
- **Augmentation:** If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z.
- **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Armstrong's Axioms are **sound** in that they generate only FDs in F^+ when applied to a set F of FDs. They are **complete** in that repeated application of these rules will generate all FDs in the closure F^+. (We will not prove these claims.) It is convenient to use some additional rules while reasoning about F^+:

- **Union:** If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
- **Decomposition:** If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.

These additional rules are not essential; their soundness can be proved using Armstrong's Axioms.

To illustrate the use of these inference rules for FDs, consider a relation schema ABC with FDs $A \rightarrow B$ and $B \rightarrow C$. A **trivial FD** is one in which the right side contains only attributes that also appear on the left side; such dependencies always hold due to reflexivity. Using reflexivity, we can generate all trivial dependencies, which are of the form:

$X \rightarrow Y$, where $Y \subseteq X$, $X \subseteq ABC$, and $Y \subseteq ABC$.

From transitivity we get $A \rightarrow C$. From augmentation we get the nontrivial dependencies:

$AC \rightarrow BC$, $AB \rightarrow AC$, $AB \rightarrow CB$.

As a second example, we use a more elaborate version of the Contracts relation:

Contracts(*contractid*, *supplierid*, *projectid*, *deptid*, *partid*, *qty*, *value*)

We denote the schema for Contracts as $CSJDPQV$. The meaning of a tuple in this relation is that the contract with *contractid* C is an agreement that supplier S (*supplierid*) will supply Q items of part P (*partid*) to project J (*projectid*) associated with department D (*deptid*); the value V of this contract is equal to *value*.

The following ICs are known to hold:

1. The contract id C is a key: $C \rightarrow CSJDPQV$.
2. A project purchases a given part using a single contract: $JP \rightarrow C$.
3. A department purchases at most one part from a supplier: $SD \rightarrow P$.

Several additional FDs hold in the closure of the set of given FDs:

From $JP \rightarrow C$, $C \rightarrow CSJDPQV$ and transitivity, we infer $JP \rightarrow CSJDPQV$.

From $SD \rightarrow P$ and augmentation, we infer $SDJ \rightarrow JP$.

From $SDJ \rightarrow JP$, $JP \rightarrow CSJDPQV$ and transitivity, we infer $SDJ \rightarrow CSJDPQV$. (Incidentally, while it may appear tempting to do so, we *cannot* conclude $SD \rightarrow CSDPQV$, canceling J on both sides. FD inference is not like arithmetic multiplication!)

We can infer several additional FDs that are in the closure by using augmentation or decomposition. For example, from $C \rightarrow CSJDPQV$, using decomposition we can infer:

$C \rightarrow C$, $C \rightarrow S$, $C \rightarrow J$, $C \rightarrow D$, etc.

Finally, we have a number of trivial FDs from the reflexivity rule.

15.4.2 Attribute Closure

If we just want to check whether a given dependency, say, $X \rightarrow Y$, is in the closure of a set F of FDs, we can do so efficiently without computing F^+. We first compute the **attribute closure** X^+ with respect to F, which is the set of attributes A such that $X \rightarrow A$ can be inferred using the Armstrong Axioms. The algorithm for computing the attribute closure of a set X of attributes is shown in Figure 15.6.

```
closure = X;
repeat until there is no change: {
        if there is an FD U → V in F such that U ⊆ closure,
            then set closure = closure ∪ V
        }
```

Figure 15.6 Computing the Attribute Closure of Attribute Set X

This algorithm can be modified to find keys by starting with set X containing a single attribute and stopping as soon as *closure* contains all attributes in the relation schema.

By varying the starting attribute and the order in which the algorithm considers FDs, we can obtain all candidate keys.

15.5 NORMAL FORMS

Given a relation schema, we need to decide whether it is a good design or whether we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several **normal forms** have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

The normal forms based on FDs are *first normal form (1NF)*, *second normal form (2NF)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*. These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is in **first normal form** if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our definition of the relational model. Although some of the newer database systems are relaxing this requirement, in this chapter we will assume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

While studying normal forms, it is important to appreciate the role played by FDs. Consider a relation schema R with attributes ABC. In the absence of any ICs, any set of ternary tuples is a legal instance and there is no potential for redundancy. On the other hand, suppose that we have the FD $A \rightarrow B$. Now if several tuples have the same A value, they must also have the same B value. This potential redundancy can be predicted using the FD information. If more detailed ICs are specified, we may be able to detect more subtle redundancies as well.

We will primarily discuss redundancy that is revealed by FD information. In Section 15.8, we discuss more sophisticated ICs called *multivalued dependencies* and *join dependencies* and normal forms based on them.

15.5.1 Boyce-Codd Normal Form

Let R be a relation schema, X be a subset of the attributes of R, and let A be an attribute of R. R is in **Boyce-Codd normal form** if for every FD $X \rightarrow A$ that holds over R, one of the following statements is true:

- $A \in X$; that is, it is a trivial FD, or
- X is a superkey.

Note that if we are given a set F of FDs, according to this definition, we must consider each dependency $X \rightarrow A$ in the closure F^+ to determine whether R is in BCNF. However, we can prove that it is sufficient to check whether the left side of each dependency in F is a superkey (by computing the attribute closure and seeing if it includes all attributes of R).

Intuitively, in a BCNF relation the only nontrivial dependencies are those in which a key determines some attribute(s). Thus, each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes. Kent puts this colorfully, if a little loosely: "Each attribute must describe [an entity or relationship identified by] the key, the whole key, and nothing but the key." If we use ovals to denote attributes or sets of attributes and draw arcs to indicate FDs, a relation in BCNF has the structure illustrated in Figure 15.7, considering just one key for simplicity. (If there are several candidate keys, each candidate key can play the role of KEY in the figure, with the other attributes being the ones not in the chosen candidate key.)

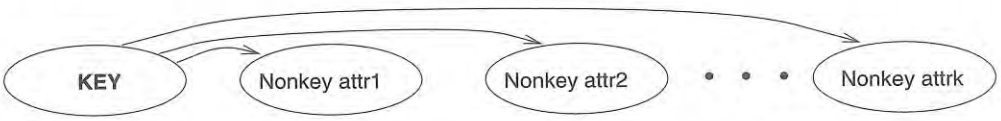


Figure 15.7 FDs in a BCNF Relation

BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form (from the point of view of redundancy) if we take into account only FD information. This point is illustrated in Figure 15.8.

X	Y	A
x	y_1	a
x	y_2	?

Figure 15.8 Instance Illustrating BCNF

This figure shows (two tuples in) an instance of a relation with three attributes X, Y, and A. There are two tuples with the same value in the X column. Now suppose that we know that this instance satisfies an FD $X \rightarrow A$. We can see that one of the tuples has the value a in the A column. What can we infer about the value in the A column in the second tuple? Using the FD, we can conclude that the second tuple also has the value a in this column. (Note that this is really the only kind of inference we can make about values in the fields of tuples by using FDs.)

But isn't this situation an example of redundancy? We appear to have stored the value a twice. Can such a situation arise in a BCNF relation? No! If this relation is

in BCNF, because A is distinct from X it follows that X must be a key. (Otherwise, the FD $X \rightarrow A$ would violate BCNF.) If X is a key, then $y_1 = y_2$, which means that the two tuples are identical. Since a relation is defined to be a *set* of tuples, we cannot have two copies of the same tuple and the situation shown in Figure 15.8 cannot arise.

Thus, if a relation is in BCNF, every field of every tuple records a piece of information that cannot be inferred (using only FDs) from the values in all other fields in (all tuples of) the relation instance.

15.5.2 Third Normal Form

Let R be a relation schema, X be a subset of the attributes of R, and A be an attribute of R. R is in **third normal form** if for every FD $X \rightarrow A$ that holds over R, one of the following statements is true:

- $A \in X$; that is, it is a trivial FD, or
- X is a superkey, or
- A is part of some key for R.

The definition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. To understand the third condition, recall that a key for a relation is a *minimal* set of attributes that uniquely determines all other attributes. A must be part of a key (any key, if there are several). It is not enough for A to be part of a superkey, because the latter condition is satisfied by each and every attribute! Finding all keys of a relation schema is known to be an NP-complete problem, and so is the problem of determining whether a relation schema is in 3NF.

Suppose that a dependency $X \rightarrow A$ causes a violation of 3NF. There are two cases:

- X *is a proper subset of some key* K. Such a dependency is sometimes called a **partial dependency**. In this case we store *(X, A)* pairs redundantly. As an example, consider the Reserves relation with attributes *SBDC* from Section 15.3.4. The only key is *SBD*, and we have the FD $S \rightarrow C$. We store the credit card number for a sailor as many times as there are reservations for that sailor.
- X *is not a proper subset of any key.* Such a dependency is sometimes called a **transitive dependency** because it means we have a chain of dependencies $K \rightarrow X \rightarrow A$. The problem is that we cannot associate an X value with a K value unless we also associate an A value with an X value. As an example, consider the Hourly_Emps relation with attributes *SNLRWH* from Section 15.3.1. The only key is S, but there is an FD $R \rightarrow W$, which gives rise to the chain $S \rightarrow R$

$\rightarrow W$. The consequence is that we cannot record the fact that employee S has rating R without knowing the hourly wage for that rating. This condition leads to insertion, deletion, and update anomalies.

Partial dependencies are illustrated in Figure 15.9, and transitive dependencies are illustrated in Figure 15.10. Note that in Figure 15.10, the set X of attributes may or may not have some attributes in common with KEY; the diagram should be interpreted as indicating only that X is not a subset of KEY.

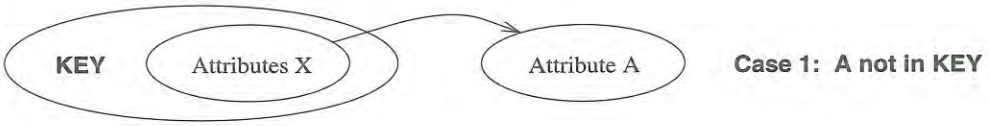


Figure 15.9 Partial Dependencies

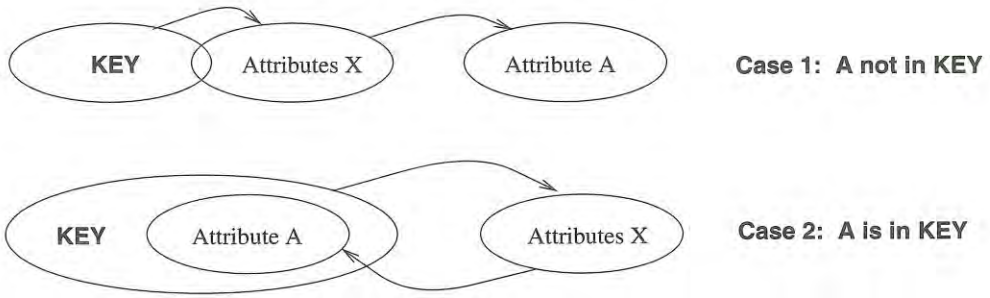


Figure 15.10 Transitive Dependencies

The motivation for 3NF is rather technical. By making an exception for certain dependencies involving key attributes, we can ensure that every relation schema can be decomposed into a collection of 3NF relations using only decompositions that have certain desirable properties (Section 15.6). Such a guarantee does not exist for BCNF relations; the 3NF definition weakens the BCNF requirements just enough to make this guarantee possible. We may therefore compromise by settling for a 3NF design. As we shall see in Chapter 16, we may sometimes accept this compromise (or even settle for a non-3NF schema) for other reasons as well.

Unlike BCNF, however, some redundancy is possible with 3NF. The problems associated with partial and transitive dependencies persist if there is a nontrivial dependency $X \rightarrow A$ and X is not a superkey, even if the relation is in 3NF because A is part of a key. To understand this point, let us revisit the Reserves relation with attributes $SBDC$ and the FD $S \rightarrow C$, which states that a sailor uses a unique credit card to pay for reservations. S is not a key, and C is not part of a key. (In fact, the only key is SBD.) Thus, this relation is not in 3NF; *(S, C)* pairs are stored redundantly. However, if we also know that credit cards uniquely identify the owner, we have the FD $C \rightarrow$

S, which means that *CBD* is also a key for Reserves. Therefore, the dependency $S \rightarrow C$ does not violate 3NF, and Reserves is in 3NF. Nonetheless, in all tuples containing the same *S* value, the same *(S, C)* pair is redundantly recorded.

For completeness, we remark that the definition of **second normal form** is essentially that partial dependencies are not allowed. Thus, if a relation is in 3NF (which precludes both partial and transitive dependencies), it is also in 2NF.

15.6 DECOMPOSITIONS

As we have seen, a relation in BCNF is free of redundancy (to be precise, redundancy that can be detected using FD information), and a relation schema in 3NF comes close. If a relation schema is not in one of these normal forms, the FDs that cause a violation can give us insight into the potential problems. The main technique for addressing such redundancy-related problems is decomposing a relation schema into relation schemas with fewer attributes.

A **decomposition of a relation schema** *R* consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of *R* and together include all attributes in *R*. Intuitively, we want to store the information in any given instance of *R* by storing projections of the instance. This section examines the use of decompositions through several examples.

We begin with the Hourly_Emps example from Section 15.3.1. This relation has attributes *SNLRWH* and two FDs: $S \rightarrow SNLRWH$ and $R \rightarrow W$. Since *R* is not a key and *W* is not part of any key, the second dependency causes a violation of 3NF.

The alternative design consisted of replacing Hourly_Emps with two relations having attributes *SNLRH* and *RW*. $S \rightarrow SNLRH$ holds over *SNLRH*, and *S* is a key. $R \rightarrow W$ holds over *RW*, and *R* is a key for *RW*. The only other dependencies that hold over these schemas are those obtained by augmentation. Thus both schemas are in BCNF.

Our decision to decompose *SNLRWH* into *SNLRH* and *RW*, rather than, say, *SNLR* and *LRWH*, was not just a good guess. It was guided by the observation that the dependency $R \rightarrow W$ caused the violation of 3NF; the most natural way to deal with this violation is to remove the attribute *W* from this schema. To compensate for removing *W* from the main schema, we can add a relation *RW*, because each *R* value is associated with at most one *W* value according to the FD $R \rightarrow W$.

A very important question must be asked at this point: If we replace a legal instance *r* of relation schema *SNLRWH* with its projections on *SNLRH* (r_1) and *RW* (r_2), can we recover *r* from r_1 and r_2? The decision to decompose *SNLRWH* into *SNLRH* and *RW* is equivalent to saying that we will store instances r_1 and r_2 instead of *r*. However,

it is the instance r that captures the intended entities or relationships. If we cannot compute r from r_1 and r_2, our attempt to deal with redundancy has effectively thrown out the baby with the bathwater. We consider this issue in more detail below.

15.6.1 Lossless-Join Decomposition

Let R be a relation schema and let F be a set of FDs over R. A decomposition of R into two schemas with attribute sets X and Y is said to be a **lossless-join decomposition with respect to F** if for every instance r of R that satisfies the dependencies in F, $\pi_X(r) \bowtie \pi_Y(r) = r$.

This definition can easily be extended to cover a decomposition of R into more than two relations. It is easy to see that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$ always holds. In general, though, the other direction does not hold. If we take projections of a relation and recombine them using natural join, we typically obtain some tuples that were not in the original relation. This situation is illustrated in Figure 15.11.

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3

Instance r

S	P
s1	p1
s2	p2
s3	p1

$\pi_{SP}(r)$

P	D
p1	d1
p2	d2
p1	d3

$\pi_{PD}(r)$

S	P	D
s1	p1	d1
s2	p2	d2
s3	p1	d3
s1	p1	d3
s3	p1	d1

$\pi_{SP}(r) \bowtie \pi_{PD}(r)$

Figure 15.11 Instances Illustrating Lossy Decompositions

By replacing the instance r shown in Figure 15.11 with the instances $\pi_{SP}(r)$ and $\pi_{PD}(r)$, we lose some information. In particular, suppose that the tuples in r denote relationships. We can no longer tell that the relationships (s_1, p_1, d_3) and (s_3, p_1, d_1) do not hold. The decomposition of schema *SPD* into *SP* and *PD* is therefore a 'lossy' decomposition if the instance r shown in the figure is legal, that is, if this instance could arise in the enterprise being modeled. (Observe the similarities between this example and the Contracts relationship set in Section 2.5.3.)

All decompositions used to eliminate redundancy **must** *be lossless.* The following simple test is very useful:

> Let R be a relation and F be a set of FDs that hold over R. The decomposition of R into relations with attribute sets R_1 and R_2 is lossless if and only if F^+ contains either the FD $R_1 \cap R_2 \rightarrow R_1$ or the FD $R_1 \cap R_2 \rightarrow R_2$.

In other words, the attributes common to R_1 and R_2 must contain a key for either R_1 or R_2. If a relation is decomposed into two relations, this test is a necessary and sufficient condition for the decomposition to be lossless-join.[2] If a relation is decomposed into more than two relations, an efficient (time polynomial in the size of the dependency set) algorithm is available to test whether or not the decomposition is lossless, but we will not discuss it.

Consider the Hourly_Emps relation again. It has attributes *SNLRWH*, and the FD $R \rightarrow W$ causes a violation of 3NF. We dealt with this violation by decomposing the relation into *SNLRH* and *RW*. Since R is common to both decomposed relations, and $R \rightarrow W$ holds, this decomposition is lossless-join.

This example illustrates a general observation:

> If an FD $X \rightarrow Y$ holds over a relation R and $X \cap Y$ is empty, the decomposition of R into $R - Y$ and XY is lossless.

X appears in both $R - Y$ (since $X \cap Y$ is empty) and XY, and it is a key for XY. Thus, the above observation follows from the test for a lossless-join decomposition.

Another important observation has to do with repeated decompositions. Suppose that a relation R is decomposed into *R1* and *R2* through a lossless-join decomposition, and that *R1* is decomposed into *R11* and *R12* through another lossless-join decomposition. Then the decomposition of R into *R11*, *R12*, and *R2* is lossless-join; by joining *R11* and *R12* we can recover *R1*, and by then joining *R1* and *R2*, we can recover R.

15.6.2 Dependency-Preserving Decomposition

Consider the Contracts relation with attributes *CSJDPQV* from Section 15.4.1. The given FDs are $C \rightarrow CSJDPQV$, $JP \rightarrow C$, and $SD \rightarrow P$. Because *SD* is not a key the dependency $SD \rightarrow P$ causes a violation of BCNF.

We can decompose Contracts into two relations with schemas *CSJDQV* and *SDP* to address this violation; the decomposition is lossless-join. There is one subtle problem, however. We can enforce the integrity constraint $JP \rightarrow C$ easily when a tuple is inserted into Contracts by ensuring that no existing tuple has the same *JP* values (as the inserted tuple) but different *C* values. Once we decompose Contracts into *CSJDQV* and *SDP*, enforcing this constraint requires an expensive join of the two relations whenever a tuple is inserted into *CSJDQV*. We say that this decomposition is not dependency-preserving.

[2] It is necessary only if we assume that only functional dependencies can be specified as integrity constraints. See Exercise 15.8.

Intuitively, a *dependency-preserving decomposition* allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple. (Note that deletions cannot cause violation of FDs.) To define dependency-preserving decompositions precisely, we have to introduce the concept of a projection of FDs.

Let R be a relation schema that is decomposed into two schemas with attribute sets X and Y, and let F be a set of FDs over R. The **projection of F on** X is the set of FDs in the closure F^+ (not just F!) that involve only attributes in X. We will denote the projection of F on attributes X as F_X. Note that a dependency $U \rightarrow V$ in F^+ is in F_X only if *all* the attributes in U and V are in X.

The decomposition of relation schema R with FDs F into schemas with attribute sets X and Y is **dependency-preserving** if $(F_X \cup F_Y)^+ = F^+$. That is, if we take the dependencies in F_X and F_Y and compute the closure of their union, we get back all dependencies in the closure of F. Therefore, we need to enforce only the dependencies in F_X and F_Y; all FDs in F^+ are then sure to be satisfied. To enforce F_X, we need to examine only relation X (on inserts to that relation). To enforce F_Y, we need to examine only relation Y.

To appreciate the need to consider the closure F^+ while computing the projection of F, suppose that a relation R with attributes ABC is decomposed into relations with attributes AB and BC. The set F of FDs over R includes $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow A$. Of these, $A \rightarrow B$ is in F_{AB} and $B \rightarrow C$ is in F_{BC}. But is this decomposition dependency-preserving? What about $C \rightarrow A$? This dependency is not implied by the dependencies listed (thus far) for F_{AB} and F_{BC}.

The closure of F contains all dependencies in F plus $A \rightarrow C$, $B \rightarrow A$, and $C \rightarrow B$. Consequently, F_{AB} also contains $B \rightarrow A$, and F_{BC} contains $C \rightarrow B$. Thus, $F_{AB} \cup F_{BC}$ contains $A \rightarrow B$, $B \rightarrow C$, $B \rightarrow A$, and $C \rightarrow B$. The closure of the dependencies in F_{AB} and F_{BC} now includes $C \rightarrow A$ (which follows from $C \rightarrow B$, $B \rightarrow A$, and transitivity). Thus, the decomposition preserves the dependency $C \rightarrow A$.

A direct application of the definition gives us a straightforward algorithm for testing whether a decomposition is dependency-preserving. (This algorithm is exponential in the size of the dependency set; a polynomial algorithm is available, although we will not discuss it.)

We began this section with an example of a lossless-join decomposition that was not dependency-preserving. Other decompositions are dependency-preserving, but not lossless. A simple example consists of a relation ABC with FD $A \rightarrow B$ that is decomposed into AB and BC.

15.7 NORMALIZATION

Having covered the concepts needed to understand the role of normal forms and decompositions in database design, we now consider algorithms for converting relations to BCNF or 3NF. If a relation schema is not in BCNF, it is possible to obtain a lossless-join decomposition into a collection of BCNF relation schemas. Unfortunately, there may not be any dependency-preserving decomposition into a collection of BCNF relation schemas. However, there is always a dependency-preserving, lossless-join decomposition into a collection of 3NF relation schemas.

15.7.1 Decomposition into BCNF

We now present an algorithm for decomposing a relation schema R into a collection of BCNF relation schemas:

1. Suppose that R is not in BCNF. Let $X \subset R$, A be a single attribute in R, and $X \rightarrow A$ be an FD that causes a violation of BCNF. Decompose R into $R - A$ and XA.
2. If either $R - A$ or XA is not in BCNF, decompose them further by a recursive application of this algorithm.

$R - A$ denotes the set of attributes other than A in R, and XA denotes the union of attributes in X and A. Since $X \rightarrow A$ violates BCNF, it is not a trivial dependency; further, A is a single attribute. Therefore, A is not in X; that is, $X \cap A$ is empty. Thus, each decomposition carried out in Step (1) is lossless-join.

The set of dependencies associated with $R - A$ and XA is the projection of F onto their attributes. If one of the new relations is not in BCNF, we decompose it further in Step (2). Since a decomposition results in relations with strictly fewer attributes, this process will terminate, leaving us with a collection of relation schemas that are all in BCNF. Further, joining instances of the (two or more) relations obtained through this algorithm will yield precisely the corresponding instance of the original relation (i.e., the decomposition into a collection of relations that are each in BCNF is a lossless-join decomposition).

Consider the Contracts relation with attributes $CSJDPQV$ and key C. We are given FDs $JP \rightarrow C$ and $SD \rightarrow P$. By using the dependency $SD \rightarrow P$ to guide the decomposition, we get the two schemas SDP and $CSJDQV$. SDP is in BCNF. Suppose that we also have the constraint that each project deals with a single supplier: $J \rightarrow S$. This means that the schema $CSJDQV$ is not in BCNF. So we decompose it further into JS and $CJDQV$. $C \rightarrow JDQV$ holds over $CJDQV$; the only other FDs that hold are those obtained from this FD by augmentation, and therefore all FDs contain a key in the left

side. Thus, each of the schemas *SDP*, *JS*, and *CJDQV* is in BCNF, and this collection of schemas also represents a lossless-join decomposition of *CSJDQV*.

The steps in this decomposition process can be visualized as a tree, as shown in Figure 15.12. The root is the original relation *CSJDPQV*, and the leaves are the BCNF relations that are the result of the decomposition algorithm, namely, *SDP*, *JS*, and *CSDQV*. Intuitively, each internal node is replaced by its children through a single decomposition step that is guided by the FD shown just below the node.

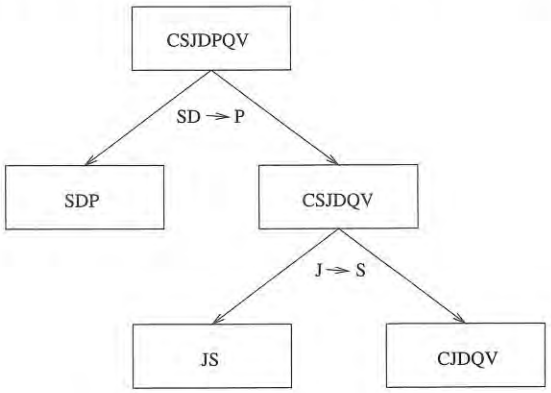


Figure 15.12 Decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV*

Redundancy in BCNF Revisited

The decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV* is not dependency-preserving. Intuitively, dependency $JP \rightarrow C$ cannot be enforced without a join. One way to deal with this situation is to add a relation with attributes *CJP*. In effect, this solution amounts to storing some information redundantly in order to make the dependency enforcement cheaper.

This is a subtle point: Each of the schemas *CJP*, *SDP*, *JS*, and *CJDQV* is in BCNF, yet there is some redundancy that can be predicted by FD information. In particular, if we join the relation instances for *SDP* and *CJDQV* and project onto the attributes *CJP*, we must get exactly the instance stored in the relation with schema *CJP*. We saw in Section 15.5.1 that there is no such redundancy within a single BCNF relation. The current example shows that redundancy can still occur across relations, even though there is no redundancy within a relation.

Alternatives in Decomposing to BCNF

Suppose that several dependencies violate BCNF. Depending on which of these dependencies we choose to guide the next decomposition step, we may arrive at quite different collections of BCNF relations. Consider Contracts. We just decomposed it into *SDP*, *JS*, and *CJDQV*. Suppose that we choose to decompose the original relation *CSJDPQV* into *JS* and *CJDPQV*, based on the FD $J \rightarrow S$. The only dependencies that hold over *CJDPQV* are $JP \rightarrow C$ and the key dependency $C \rightarrow CJDPQV$. Since *JP* is a key, *CJDPQV* is in BCNF. Thus, the schemas *JS* and *CJDPQV* represent a lossless-join decomposition of Contracts into BCNF relations.

The lesson to be learned here is that the theory of dependencies can tell us when there is redundancy and give us clues about possible decompositions to address the problem, but it cannot discriminate between decomposition alternatives. A designer has to consider the alternatives and choose one based on the semantics of the application.

BCNF and Dependency-Preservation

Sometimes, there simply is no decomposition into BCNF that is dependency-preserving. As an example, consider the relation schema *SBD*, in which a tuple denotes that sailor *S* has reserved boat *B* on date *D*. If we have the FDs $SB \rightarrow D$ (a sailor can reserve a given boat for at most one day) and $D \rightarrow B$ (on any given day at most one boat can be reserved), *SBD* is not in BCNF because *D* is not a key. If we try to decompose it, however, we cannot preserve the dependency $SB \rightarrow D$.

15.7.2 Decomposition into 3NF *

Clearly, the approach that we outlined for lossless-join decomposition into BCNF will also give us a lossless-join decomposition into 3NF. (Typically, we can stop a little earlier if we are satisfied with a collection of 3NF relations.) But this approach does not ensure dependency-preservation.

A simple modification, however, yields a decomposition into 3NF relations that is lossless-join and dependency-preserving. Before we describe this modification, we need to introduce the concept of a minimal cover for a set of FDs.

Minimal Cover for a Set of FDs

A **minimal cover** for a set F of FDs is a set G of FDs such that:

1. Every dependency in G is of the form $X \rightarrow A$, where A is a single attribute.
2. The closure F^+ is equal to the closure G^+.

3. If we obtain a set H of dependencies from G by deleting one or more dependencies, or by deleting attributes from a dependency in G, then $F^+ \neq H^+$.

Intuitively, a minimal cover for a set F of FDs is an equivalent set of dependencies that is *minimal* in two respects: (1) Every dependency is as small as possible; that is, each attribute on the left side is necessary and the right side is a single attribute. (2) Every dependency in it is required in order for the closure to be equal to F^+.

As an example, let F be the set of dependencies:

$$A \rightarrow B,\ ABCD \rightarrow E,\ EF \rightarrow G,\ EF \rightarrow H, \text{ and } ACDF \rightarrow EG.$$

First, let us rewrite $ACDF \rightarrow EG$ so that every right side is a single attribute:

$$ACDF \rightarrow E \text{ and } ACDF \rightarrow G.$$

Next consider $ACDF \rightarrow G$. This dependency is implied by the following FDs:

$$A \rightarrow B,\ ABCD \rightarrow E, \text{ and } EF \rightarrow G.$$

Therefore, we can delete it. Similarly, we can delete $ACDF \rightarrow E$. Next consider $ABCD \rightarrow E$. Since $A \rightarrow B$ holds, we can replace it with $ACD \rightarrow E$. (At this point, the reader should verify that each remaining FD is minimal and required.) Thus, a minimal cover for F is the set:

$$A \rightarrow B,\ ACD \rightarrow E,\ EF \rightarrow G, \text{ and } EF \rightarrow H.$$

The preceding example suggests a general algorithm for obtaining a minimal cover of a set F of FDs:

1. **Put the FDs in a standard form:** Obtain a collection G of equivalent FDs with a single attribute on the right side (using the decomposition axiom).
2. **Minimize the left side of each FD:** For each FD in G, check each attribute in the left side to see if it can be deleted while preserving equivalence to F^+.
3. **Delete redundant FDs:** Check each remaining FD in G to see if it can be deleted while preserving equivalence to F^+.

Note that the order in which we consider FDs while applying these steps could produce different minimal covers; there could be several minimal covers for a given set of FDs.

More important, it is necessary to minimize the left sides of FDs *before* checking for redundant FDs. If these two steps are reversed, the final set of FDs could still contain some redundant FDs (i.e., not be a minimal cover), as the following example illustrates. Let F be the set of dependencies, each of which is already in the standard form:

$ABCD \rightarrow E$, $E \rightarrow D$, $A \rightarrow B$, and $AC \rightarrow D$.

Observe that none of these FDs is redundant; if we checked for redundant FDs first, we would get the same set of FDs F. The left side of $ABCD \rightarrow E$ can be replaced by AC while preserving equivalence to F^+, and we would stop here if we checked for redundant FDs in F before minimizing the left sides. However, the set of FDs we have is not a minimal cover:

$AC \rightarrow E$, $E \rightarrow D$, $A \rightarrow B$, and $AC \rightarrow D$.

From transitivity, the first two FDs imply the last FD, which can therefore be deleted while preserving equivalence to F^+. The important point to note is that $AC \rightarrow D$ becomes redundant only after we replace $ABCD \rightarrow E$ with $AC \rightarrow E$. If we minimize left sides of FDs first and then check for redundant FDs, we are left with the first three FDs in the preceding list, which is indeed a minimal cover for F.

Dependency-Preserving Decomposition into 3NF

Returning to the problem of obtaining a lossless-join, dependency-preserving decomposition into 3NF relations, let R be a relation with a set F of FDs that is a minimal cover, and let R_1, R_2, ..., R_n be a lossless-join decomposition of R. For $1 \leq i \leq n$, suppose that each R_i is in 3NF and let F_i denote the projection of F onto the attributes of R_i. Do the following:

- Identify the set N of dependencies in F that are not **preserved**, that is, not included in the closure of the union of F_is.
- For each FD $X \rightarrow A$ in N, create a relation schema XA and add it to the decomposition of R.

Obviously, every dependency in F is preserved if we replace R by the R_is plus the schemas of the form XA added in this step. The R_is are given to be in 3NF. We can show that each of the schemas XA is in 3NF as follows: Since $X \rightarrow A$ is in the minimal cover F, $Y \rightarrow A$ does not hold for any Y that is a strict subset of X. Therefore, X is a key for XA. Further, if any other dependencies hold over XA, the right side can involve only attributes in X because A is a single attribute (because $X \rightarrow A$ is an FD in a minimal cover). Since X is a key for XA, none of these additional dependencies causes a violation of 3NF (although they might cause a violation of BCNF).

As an optimization, if the set N contains several FDs with the same left side, say, $X \rightarrow A_1$, $X \rightarrow A_2$, ..., $X \rightarrow A_n$, we can replace them with a single equivalent FD $X \rightarrow A_1 \ldots A_n$. Therefore, we produce one relation schema $XA_1 \ldots A_n$, instead of several schemas $XA_1, \ldots, XA_n$, which is generally preferable.

Consider the Contracts relation with attributes $CSJDPQV$ and FDs $JP \rightarrow C$, $SD \rightarrow P$, and $J \rightarrow S$. If we decompose $CSJDPQV$ into SDP and $CSJDQV$, then SDP is in BCNF, but $CSJDQV$ is not even in 3NF. So we decompose it further into JS and $CJDQV$. The relation schemas SDP, JS, and $CJDQV$ are in 3NF (in fact, in BCNF), and the decomposition is lossless-join. However, the dependency $JP \rightarrow C$ is not preserved. This problem can be addressed by adding a relation schema CJP to the decomposition.

3NF Synthesis

We have assumed that the design process starts with an ER diagram, and that our use of FDs is primarily to guide decisions about decomposition. The algorithm for obtaining a lossless-join, dependency-preserving decomposition was presented in the previous section from this perspective—a lossless-join decomposition into 3NF is straightforward, and the algorithm addresses dependency-preservation by adding extra relation schemas.

An alternative approach, called **synthesis**, is to take all the attributes over the original relation R and a minimal cover F for the FDs that hold over it, and to add a relation schema XA to the decomposition of R for each FD $X \rightarrow A$ in F.

The resulting collection of relation schemas is in 3NF and preserves all FDs. If it is not a lossless-join decomposition of R, we can make it so by adding a relation schema that contains just those attributes that appear in some key. This algorithm gives us a lossless-join, dependency-preserving decomposition into 3NF, and has polynomial complexity—polynomial algorithms are available for computing minimal covers, and a key can be found in polynomial time (even though finding all keys is known to be NP-complete). The existence of a polynomial algorithm for obtaining a lossless-join, dependency-preserving decomposition into 3NF is surprising when we consider that testing whether a given schema is in 3NF is NP-complete.

As an example, consider a relation ABC with FDs $F = \{A \rightarrow B,\ C \rightarrow B\}$. The first step yields the relation schemas AB and BC. This is not a lossless-join decomposition of ABC; $AB \cap BC$ is B, and neither $B \rightarrow A$ nor $B \rightarrow C$ is in F^+. If we add a schema AC, we have the lossless-join property as well. Although the collection of relations AB, BC, and AC is a dependency-preserving, lossless-join decomposition of ABC, we obtained it through a process of *synthesis*, rather than through a process of repeated decomposition. We note that the decomposition produced by the synthesis approach is heavily dependent on the minimal cover that is used.

As another example of the synthesis approach, consider the Contracts relation with attributes $CSJDPQV$ and the following FDs:

$$C \rightarrow CSJDPQV,\ JP \rightarrow C,\ SD \rightarrow P, \text{ and } J \rightarrow S.$$

This set of FDs is not a minimal cover, and so we must find one. We first replace $C \rightarrow CSJDPQV$ with the FDs:

$$C \rightarrow S,\ C \rightarrow J,\ C \rightarrow D,\ C \rightarrow P,\ C \rightarrow Q, \text{ and } C \rightarrow V.$$

The FD $C \rightarrow P$ is implied by $C \rightarrow S$, $C \rightarrow D$, and $SD \rightarrow P$; so we can delete it. The FD $C \rightarrow S$ is implied by $C \rightarrow J$ and $J \rightarrow S$; so we can delete it. This leaves us with a minimal cover:

$$C \rightarrow J,\ C \rightarrow D,\ C \rightarrow Q,\ C \rightarrow V,\ JP \rightarrow C,\ SD \rightarrow P, \text{ and } J \rightarrow S.$$

Using the algorithm for ensuring dependency-preservation, we obtain the relational schema CJ, CD, CQ, CV, CJP, SDP, and JS. We can improve this schema by combining relations for which C is the key into $CDJPQV$. In addition, we have SDP and JS in our decomposition. Since one of these relations ($CDJPQV$) is a superkey, we are done.

Comparing this decomposition with the one that we obtained earlier in this section, we find that they are quite close, with the only difference being that one of them has $CDJPQV$ instead of CJP and $CJDQV$. In general, however, there could be significant differences.

Database designers typically use a conceptual design methodology (e.g., ER design) to arrive at an initial database design. Given this, the approach of repeated decompositions to rectify instances of redundancy is likely to be the most natural use of FDs and normalization techniques. However, a designer can also consider the alternative designs suggested by the synthesis approach.

15.8 OTHER KINDS OF DEPENDENCIES *

FDs are probably the most common and important kind of constraint from the point of view of database design. However, there are several other kinds of dependencies. In particular, there is a well-developed theory for database design using *multivalued dependencies* and *join dependencies*. By taking such dependencies into account, we can identify potential redundancy problems that cannot be detected using FDs alone.

This section illustrates the kinds of redundancy that can be detected using multivalued dependencies. Our main observation, however, is that simple guidelines (which can be checked using only FD reasoning) can tell us whether we even need to worry about complex constraints such as multivalued and join dependencies. We also comment on the role of *inclusion dependencies* in database design.

15.8.1 Multivalued Dependencies

Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*. The meaning of a tuple is that teacher T can teach course C, and book B is a recommended text for the course. There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the instructor. The instance shown in Figure 15.13 illustrates this situation.

course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

Figure 15.13 BCNF Relation with Redundancy That Is Revealed by MVDs

There are three points to note here:

- The relation schema *CTB* is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over *CTB*.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.

The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be expressed in terms of FDs. This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes *CT* and Text with attributes *CB*. Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate. (See Section 2.5.3 for a further discussion of ternary versus binary relationships.) Given the subjectivity of ER design, however, we might create a ternary relationship. A careful analysis of the MVD information would then reveal the problem.

Let R be a relation schema and let X and Y be subsets of the attributes of R. Intuitively, the **multivalued dependency** $X \rightarrow\rightarrow Y$ is said to hold over R if, in every legal

instance r of R, each X value is associated with a set of Y values and this set is independent of the values in the other attributes.

Formally, if the MVD $X \rightarrow\rightarrow Y$ holds over R and $Z = R - XY$, the following must be true for every legal instance r of R:

> If $t_1 \in r$, $t_2 \in r$ and $t_1.X = t_2.X$, then there must be some $t_3 \in r$ such that $t_1.XY = t_3.XY$ and $t_2.Z = t_3.Z$.

Figure 15.14 illustrates this definition. If we are given the first two tuples and told that the MVD $X \rightarrow\rightarrow Y$ holds over this relation, we can infer that the relation instance must also contain the third tuple. Indeed, by interchanging the roles of the first two tuples—treating the first tuple as t_2 and the second tuple as t_1—we can deduce that the tuple t_4 must also be in the relation instance.

X	Y	Z	
a	b_1	c_1	— tuple t_1
a	b_2	c_2	— tuple t_2
a	b_1	c_2	— tuple t_3
a	b_2	c_1	— tuple t_4

Figure 15.14 Illustration of MVD Definition

This table suggests another way to think about MVDs: If $X \rightarrow\rightarrow Y$ holds over R, then $\pi_{YZ}(\sigma_{X=x}(R)) = \pi_Y(\sigma_{X=x}(R)) \times \pi_Z(\sigma_{X=x}(R))$ in every legal instance of R, for any value x that appears in the X column of R. In other words, consider groups of tuples in R with the same X-value, for each X-value. In each such group consider the projection onto the attributes YZ. This projection must be equal to the cross-product of the projections onto Y and Z. That is, for a given X-value, the Y-values and Z-values are independent. (From this definition it is easy to see that $X \rightarrow\rightarrow Y$ must hold whenever $X \rightarrow Y$ holds. If the FD $X \rightarrow Y$ holds, there is exactly one Y-value for a given X-value, and the conditions in the MVD definition hold trivially. The converse does not hold, as Figure 15.14 illustrates.)

Returning to our *CTB* example, the constraint that course texts are independent of instructors can be expressed as $C \rightarrow\rightarrow T$. In terms of the definition of MVDs, this constraint can be read as follows:

> "If (there is a tuple showing that) C is taught by teacher T,
> and (there is a tuple showing that) C has book B as text,
> then (there is a tuple showing that) C is taught by T and has text B.

Given a set of FDs and MVDs, in general we can infer that several additional FDs and MVDs hold. A sound and complete set of inference rules consists of the three Armstrong Axioms plus five additional rules. Three of the additional rules involve only MVDs:

- **MVD Complementation:** If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow R - XY$.
- **MVD Augmentation:** If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$.
- **MVD Transitivity:** If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$.

As an example of the use of these rules, since we have $C \twoheadrightarrow T$ over CTB, MVD complementation allows us to infer that $C \twoheadrightarrow CTB - CT$ as well, that is, $C \twoheadrightarrow B$. The remaining two rules relate FDs and MVDs:

- **Replication:** If $X \rightarrow Y$, then $X \twoheadrightarrow Y$.
- **Coalescence:** If $X \twoheadrightarrow Y$ and there is a W such that $W \cap Y$ is empty, $W \rightarrow Z$, and $Y \supseteq Z$, then $X \rightarrow Z$.

Observe that replication states that every FD is also an MVD.

15.8.2 Fourth Normal Form

Fourth normal form is a direct generalization of BCNF. Let R be a relation schema, X and Y be nonempty subsets of the attributes of R, and F be a set of dependencies that includes both FDs and MVDs. R is said to be in **fourth normal form (4NF)** if for every MVD $X \twoheadrightarrow Y$ that holds over R, one of the following statements is true:

- $Y \subseteq X$ or $XY = R$, or
- X is a superkey.

In reading this definition, it is important to understand that the definition of a key has not changed—the key must uniquely determine all attributes through FDs alone. $X \twoheadrightarrow Y$ is a **trivial MVD** if $Y \subseteq X \subseteq R$ or $XY = R$; such MVDs always hold.

The relation CTB is not in 4NF because $C \twoheadrightarrow T$ is a nontrivial MVD and C is not a key. We can eliminate the resulting redundancy by decomposing CTB into CT and CB; each of these relations is then in 4NF.

To use MVD information fully, we must understand the theory of MVDs. However, the following result due to Date and Fagin identifies conditions—detected using only FD information!—under which we can safely ignore MVD information. That is, using MVD information in addition to the FD information will not reveal any redundancy. Therefore, if these conditions hold, we do not even need to identify all MVDs.

If a relation schema is in BCNF, and at least one of its keys consists of a single attribute, it is also in 4NF.

An important assumption is implicit in any application of the preceding result: *The set of FDs identified thus far is indeed the set of all FDs that hold over the relation.* This assumption is important because the result relies on the relation being in BCNF, which in turn depends on the set of FDs that hold over the relation.

We illustrate this point using an example. Consider a relation schema $ABCD$ and suppose that the FD $A \rightarrow BCD$ and the MVD $B \rightarrow\rightarrow C$ are given. Considering only these dependencies, this relation schema appears to be a counter example to the result. The relation has a simple key, appears to be in BCNF, and yet is not in 4NF because $B \rightarrow\rightarrow C$ causes a violation of the 4NF conditions. But let's take a closer look.

Figure 15.15 shows three tuples from an instance of $ABCD$ that satisfies the given MVD $B \rightarrow\rightarrow C$. From the definition of an MVD, given tuples t_1 and t_2, it follows

B	C	A	D	
b	c_1	a_1	d_1	— tuple t_1
b	c_2	a_2	d_2	— tuple t_2
b	c_1	a_2	d_2	— tuple t_3

Figure 15.15 Three Tuples from a Legal Instance of $ABCD$

that tuple t_3 must also be included in the instance. Consider tuples t_2 and t_3. From the given FD $A \rightarrow BCD$ and the fact that these tuples have the same A-value, we can deduce that $c_1 = c_2$. Thus, we see that the FD $B \rightarrow C$ must hold over $ABCD$ whenever the FD $A \rightarrow BCD$ and the MVD $B \rightarrow\rightarrow C$ hold. If $B \rightarrow C$ holds, the relation $ABCD$ is not in BCNF (unless additional FDs hold that make B a key)!

Thus, the apparent counter example is really not a counter example—rather, it illustrates the importance of correctly identifying all FDs that hold over a relation. In this example $A \rightarrow BCD$ is not the only FD; the FD $B \rightarrow C$ also holds but was not identified initially. Given a set of FDs and MVDs, the inference rules can be used to infer additional FDs (and MVDs); to apply the Date-Fagin result without first using the MVD inference rules, we must be certain that we have identified all the FDs.

In summary, the Date-Fagin result offers a convenient way to check that a relation is in 4NF (without reasoning about MVDs) if we are confident that we have identified all FDs. At this point the reader is invited to go over the examples we have discussed in this chapter and see if there is a relation that is not in 4NF.

15.8.3 Join Dependencies

A join dependency is a further generalization of MVDs. A **join dependency** (JD) $\bowtie \{R_1, \ldots, R_n\}$ is said to hold over a relation R if $R_1, \ldots, R_n$ is a lossless-join decomposition of R.

An MVD $X \rightarrow\rightarrow Y$ over a relation R can be expressed as the join dependency $\bowtie \{XY, X(R-Y)\}$. As an example, in the *CTB* relation, the MVD $C \rightarrow\rightarrow T$ can be expressed as the join dependency $\bowtie \{CT, CB\}$.

Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

15.8.4 Fifth Normal Form

A relation schema R is said to be in **fifth normal form (5NF)** if for every JD $\bowtie \{R_1, \ldots, R_n\}$ that holds over R, one of the following statements is true:

- $R_i = R$ for some i, or
- The JD is implied by the set of those FDs over R in which the left side is a key for R.

The second condition deserves some explanation, since we have not presented inference rules for FDs and JDs taken together. Intuitively, we must be able to show that the decomposition of R into $\{R_1, \ldots, R_n\}$ is lossless-join whenever the **key dependencies** (FDs in which the left side is a key for R) hold. $\bowtie \{R_1, \ldots, R_n\}$ is a **trivial JD** if $R_i = R$ for some i; such a JD always holds.

The following result, also due to Date and Fagin, identifies conditions—again, detected using only FD information—under which we can safely ignore JD information.

> If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also in 5NF.

The conditions identified in this result are sufficient for a relation to be in 5NF, but not necessary. The result can be very useful in practice because it allows us to conclude that a relation is in 5NF *without ever identifying the MVDs and JDs that may hold over the relation.*

15.8.5 Inclusion Dependencies

MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about. In contrast,

inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design (beyond the ER design stage).

Informally, an inclusion dependency is a statement of the form that some columns of a relation are contained in other columns (usually of a second relation). A foreign key constraint is an example of an inclusion dependency; the referring column(s) in one relation must be contained in the primary key column(s) of the referenced relation. As another example, if R and S are two relations obtained by translating two entity sets such that every R entity is also an S entity, we would have an inclusion dependency; projecting R on its key attributes yields a relation that is contained in the relation obtained by projecting S on its key attributes.

The main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency. For example, if we have an inclusion dependency $AB \subseteq CD$, while decomposing the relation schema containing AB, we should ensure that at least one of the schemas obtained in the decomposition contains both A and B. Otherwise, we cannot check the inclusion dependency $AB \subseteq CD$ without reconstructing the relation containing AB.

Most inclusion dependencies in practice are *key-based*, that is, involve only keys. Foreign key constraints are a good example of key-based inclusion dependencies. An ER diagram that involves ISA hierarchies also leads to key-based inclusion dependencies. If all inclusion dependencies are key-based, we rarely have to worry about splitting attribute groups that participate in inclusions, since decompositions usually do not split the primary key. Note, however, that going from 3NF to BCNF always involves splitting some key (hopefully not the primary key!), since the dependency guiding the split is of the form $X \rightarrow A$ where A is part of a key.

15.9 POINTS TO REVIEW

- *Redundancy*, storing the same information several times in a database, can result in *update anomalies* (all copies need to be updated), *insertion anomalies* (certain information cannot be stored unless other information is stored as well), and *deletion anomalies* (deleting some information means loss of other information as well). We can reduce redundancy by replacing a relation schema R with several smaller relation schemas. This process is called *decomposition*. **(Section 15.1)**
- A *functional dependency* $X \rightarrow Y$ is a type of IC. It says that if two tuples agree upon (i.e., have the same) values in the X attributes, then they also agree upon the values in the Y attributes. **(Section 15.2)**
- FDs can help to refine subjective decisions made during conceptual design. **(Section 15.3)**

- An FD f *is implied by* a set F of FDs if for all relation instances where F holds, f also holds. The closure of a set F of FDs is the set of all FDs F^+ implied by F. *Armstrong's Axioms* are a sound and complete set of rules to generate all FDs in the closure. An FD $X \rightarrow Y$ is *trivial* if X contains only attributes that also appear in Y. The *attribute closure* X^+ of a set of attributes X with respect to a set of FDs F is the set of attributes A such that $X \rightarrow A$ can be inferred using Armstrong's Axioms. **(Section 15.4)**

- A *normal form* is a property of a relation schema indicating the type of redundancy that the relation schema exhibits. If a relation schema is in *Boyce-Codd normal form* (BCNF), then the only nontrivial FDs are key constraints. If a relation is in third normal form (3NF), then all nontrivial FDs are key constraints or their right side is part of a candidate key. Thus, every relation that is in BCNF is also in 3NF, but not vice versa. **(Section 15.5)**

- A decomposition of a relation schema R into two relation schemas X and Y is a *lossless-join* decomposition with respect to a set of FDs F if for any instance r of R that satisfies the FDs in F, $\pi_X(r) \bowtie \pi_Y(r) = r$. The decomposition of R into X and Y is lossless-join if and only if F^+ contains either $X \cap Y \rightarrow X$ or the FD $X \cap Y \rightarrow Y$. The decomposition is *dependency-preserving* if we can enforce all FDs that are given to hold on R by simply enforcing FDs on X and FDs on Y independently (i.e., without joining X and Y). **(Section 15.6)**

- There is an algorithm to obtain a lossless-join decomposition of a relation into a collection of BCNF relation schemas, but sometimes there is no dependency-preserving decomposition into BCNF schemas. We also discussed an algorithm for decomposing a relation schema into a collection of 3NF relation schemas. There is always a lossless-join, dependency-preserving decomposition into a collection of 3NF relation schemas. A *minimal cover* of a set of FDs is an equivalent set of FDs that has certain minimality properties (intuitively, the set of FDs is as small as possible). Instead of decomposing a relation schema, we can also *synthesize* a corresponding collection of 3NF relation schemas. **(Section 15.7)**

- Other kinds of dependencies include *multivalued dependencies*, *join dependencies*, and *inclusion dependencies*. Fourth and fifth normal forms are more stringent than BCNF, and eliminate redundancy due to multivalued and join dependencies, respectively. **(Section 15.8)**

EXERCISES

Exercise 15.1 Briefly answer the following questions.

1. Define the term *functional dependency*.
2. Give a set of FDs for the relation schema $R(A,B,C,D)$ with primary key AB under which R is in 1NF but not in 2NF.

3. Give a set of FDs for the relation schema $R(A,B,C,D)$ with primary key AB under which R is in 2NF but not in 3NF.
4. Consider the relation schema $R(A,B,C)$, which has the FD $B \rightarrow C$. If A is a candidate key for R, is it possible for R to be in BCNF? If so, under what conditions? If not, explain why not.
5. Suppose that we have a relation schema $R(A,B,C)$ representing a relationship between two entity sets with keys A and B, respectively, and suppose that R has (among others) the FDs $A \rightarrow B$ and $B \rightarrow A$. Explain what such a pair of dependencies means (i.e., what they imply about the relationship that the relation models).

Exercise 15.2 Consider a relation R with five attributes $ABCDE$. You are given the following dependencies: $A \rightarrow B$, $BC \rightarrow E$, and $ED \rightarrow A$.

1. List all keys for R.
2. Is R in 3NF?
3. Is R in BCNF?

Exercise 15.3 Consider the following collection of relations and dependencies. Assume that each relation is obtained through decomposition from a relation with attributes $ABCDEFGHI$ and that all the known dependencies over relation $ABCDEFGHI$ are listed for each question. (The questions are independent of each other, obviously, since the given dependencies over $ABCDEFGHI$ are different.) For each (sub) relation: (a) State the strongest normal form that the relation is in. (b) If it is not in BCNF, decompose it into a collection of BCNF relations.

1. $R1(A,C,B,D,E)$, $A \rightarrow B$, $C \rightarrow D$
2. $R2(A,B,F)$, $AC \rightarrow E$, $B \rightarrow F$
3. $R3(A,D)$, $D \rightarrow G$, $G \rightarrow H$
4. $R4(D,C,H,G)$, $A \rightarrow I$, $I \rightarrow A$
5. $R5(A,I,C,E)$

Exercise 15.4 Suppose that we have the following three tuples in a legal instance of a relation schema S with three attributes ABC (listed in order): (1,2,3), (4,2,3), and (5,3,3).

1. Which of the following dependencies can you infer does *not* hold over schema S?

   (a) $A \rightarrow B$ (b) $BC \rightarrow A$ (c) $B \rightarrow C$

2. Can you identify any dependencies that hold over S?

Exercise 15.5 Suppose you are given a relation R with four attributes, $ABCD$. For each of the following sets of FDs, assuming those are the only dependencies that hold for R, do the following: (a) Identify the candidate key(s) for R. (b) Identify the best normal form that R satisfies (1NF, 2NF, 3NF, or BCNF). (c) If R is not in BCNF, decompose it into a set of BCNF relations that preserve the dependencies.

1. $C \rightarrow D$, $C \rightarrow A$, $B \rightarrow C$

2. $B \rightarrow C$, $D \rightarrow A$
3. $ABC \rightarrow D$, $D \rightarrow A$
4. $A \rightarrow B$, $BC \rightarrow D$, $A \rightarrow C$
5. $AB \rightarrow C$, $AB \rightarrow D$, $C \rightarrow A$, $D \rightarrow B$

Exercise 15.6 Consider the attribute set $R = ABCDEGH$ and the FD set $F = \{AB \rightarrow C, AC \rightarrow B, AD \rightarrow E, B \rightarrow D, BC \rightarrow A, E \rightarrow G\}$.

1. For each of the following attribute sets, do the following: (i) Compute the set of dependencies that hold over the set and write down a minimal cover. (ii) Name the strongest normal form that is not violated by the relation containing these attributes. (iii) Decompose it into a collection of BCNF relations if it is not in BCNF.

   (a) ABC (b) $ABCD$ (c) $ABCEG$ (d) $DCEGH$ (e) $ACEH$

2. Which of the following decompositions of $R = ABCDEG$, with the same set of dependencies F, is (a) dependency-preserving? (b) lossless-join?

   (a) $\{AB, BC, ABDE, EG\}$

   (b) $\{ABC, ACDE, ADG\}$

Exercise 15.7 Let R be decomposed into $R_1, R_2, \ldots, R_n$. Let F be a set of FDs on R.

1. Define what it means for F to *be preserved* in the set of decomposed relations.
2. Describe a polynomial-time algorithm to test dependency-preservation.
3. Projecting the FDs stated over a set of attributes X onto a subset of attributes Y requires that we consider the closure of the FDs. Give an example where considering the closure is important in testing dependency-preservation; that is, considering just the given FDs gives incorrect results.

Exercise 15.8 Consider a relation R that has three attributes ABC. It is decomposed into relations R_1 with attributes AB and R_2 with attributes BC.

1. State the definition of a lossless-join decomposition with respect to this example. Answer this question concisely by writing a relational algebra equation involving R, R_1, and R_2.
2. Suppose that $B \rightarrow\rightarrow C$. Is the decomposition of R into R_1 and R_2 lossless-join? Reconcile your answer with the observation that neither of the FDs $R_1 \cap R_2 \rightarrow R_1$ nor $R_1 \cap R_2 \rightarrow R_2$ hold, in light of the simple test offering a necessary and sufficient condition for lossless-join decomposition into two relations in Section 15.6.1.
3. If you are given the following instances of R_1 and R_2, what can you say about the instance of R from which these were obtained? Answer this question by listing tuples that are definitely in R and listing tuples that are possibly in R.

   Instance of $R_1 = \{(5,1), (6,1)\}$
   Instance of $R_2 = \{(1,8), (1,9)\}$

   Can you say that attribute B definitely *is* or *is not* a key for R?

Exercise 15.9 Suppose you are given a relation $R(A,B,C,D)$. For each of the following sets of FDs, assuming they are the only dependencies that hold for R, do the following: (a) Identify the candidate key(s) for R. (b) State whether or not the proposed decomposition of R into smaller relations is a good decomposition, and briefly explain why or why not.

1. $B \rightarrow C$, $D \rightarrow A$; decompose into BC and AD.
2. $AB \rightarrow C$, $C \rightarrow A$, $C \rightarrow D$; decompose into ACD and BC.
3. $A \rightarrow BC$, $C \rightarrow AD$; decompose into ABC and AD.
4. $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$; decompose into AB and ACD.
5. $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$; decompose into AB, AD and CD.

Exercise 15.10 Suppose that we have the following four tuples in a relation S with three attributes ABC: (1,2,3), (4,2,3), (5,3,3), (5,3,4). Which of the following functional ($\rightarrow$) and multivalued ($\rightarrow\rightarrow$) dependencies can you infer does *not* hold over relation S?

1. $A \rightarrow B$
2. $A \rightarrow\rightarrow B$
3. $BC \rightarrow A$
4. $BC \rightarrow\rightarrow A$
5. $B \rightarrow C$
6. $B \rightarrow\rightarrow C$

Exercise 15.11 Consider a relation R with five attributes $ABCDE$.

1. For each of the following instances of R, state whether (a) it violates the FD $BC \rightarrow D$, and (b) it violates the MVD $BC \rightarrow\rightarrow D$:
   (a) { } (i.e., empty relation)
   (b) {(a,2,3,4,5), (2,a,3,5,5)}
   (c) {(a,2,3,4,5), (2,a,3,5,5), (a,2,3,4,6)}
   (d) {(a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5)}
   (e) {(a,2,3,4,5), (2,a,3,7,5), (a,2,3,4,6)}
   (f) {(a,2,3,4,5), (2,a,3,4,5), (a,2,3,6,5), (a,2,3,6,6)}
   (g) {(a,2,3,4,5), (a,2,3,6,5), (a,2,3,6,6), (a,2,3,4,6)}
2. If each instance for R listed above is legal, what can you say about the FD $A \rightarrow B$?

Exercise 15.12 JDs are motivated by the fact that sometimes a relation that cannot be decomposed into two smaller relations in a lossless-join manner can be so decomposed into three or more relations. An example is a relation with attributes *supplier*, *part*, and *project*, denoted SPJ, with no FDs or MVDs. The JD $\bowtie \{SP, PJ, JS\}$ holds.

From the JD, the set of relation schemes SP, PJ, and JS is a lossless-join decomposition of SPJ. Construct an instance of SPJ to illustrate that no two of these schemes suffice.

Exercise 15.13 Consider a relation R with attributes $ABCDE$. Let the following FDs be given: $A \rightarrow BC$, $BC \rightarrow E$, and $E \rightarrow DA$. Similarly, let S be a relation with attributes $ABCDE$ and let the following FDs be given: $A \rightarrow BC$, $B \rightarrow E$, and $E \rightarrow DA$. (Only the second dependency differs from those that hold over R.) You do not know whether or which other (join) dependencies hold.

1. Is R in BCNF?
2. Is R in 4NF?
3. Is R in 5NF?
4. Is S in BCNF?
5. Is S in 4NF?
6. Is S in 5NF?

Exercise 15.14 Let us say that an FD $X \rightarrow Y$ is *simple* if Y is a single attribute.

1. Replace the FD $AB \rightarrow CD$ by the smallest equivalent collection of simple FDs.
2. Prove that every FD $X \rightarrow Y$ in a set of FDs F can be replaced by a set of simple FDs such that F^+ is equal to the closure of the new set of FDs.

Exercise 15.15 Prove that Armstrong's Axioms are sound and complete for FD inference. That is, show that repeated application of these axioms on a set F of FDs produces exactly the dependencies in F^+.

Exercise 15.16 Describe a linear-time (in the size of the set of FDs, where the size of each FD is the number of attributes involved) algorithm for finding the attribute closure of a set of attributes with respect to a set of FDs.

Exercise 15.17 Consider a scheme R with FDs F that is decomposed into schemes with attributes X and Y. Show that this is dependency-preserving if $F \subseteq (F_X \cup F_Y)^+$.

Exercise 15.18 Let R be a relation schema with a set F of FDs. Prove that the decomposition of R into R_1 and R_2 is lossless-join if and only if F^+ contains $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$.

Exercise 15.19 Prove that the optimization of the algorithm for lossless-join, dependency-preserving decomposition into 3NF relations (Section 15.7.2) is correct.

Exercise 15.20 Prove that the 3NF synthesis algorithm produces a lossless-join decomposition of the relation containing all the original attributes.

Exercise 15.21 Prove that an MVD $X \rightarrow\rightarrow Y$ over a relation R can be expressed as the join dependency $\bowtie \{XY, X(R-Y)\}$.

Exercise 15.22 Prove that if R has only one key, it is in BCNF if and only if it is in 3NF.

Exercise 15.23 Prove that if R is in 3NF and every key is simple, then R is in BCNF.

Exercise 15.24 Prove these statements:

1. If a relation scheme is in BCNF and at least one of its keys consists of a single attribute, it is also in 4NF.
2. If a relation scheme is in 3NF and each key has a single attribute, it is also in 5NF.

Exercise 15.25 Give an algorithm for testing whether a relation scheme is in BCNF. The algorithm should be polynomial in the size of the set of given FDs. (The *size* is the sum over all FDs of the number of attributes that appear in the FD.) Is there a polynomial algorithm for testing whether a relation scheme is in 3NF?

PROJECT-BASED EXERCISES

Exercise 15.26 Minibase provides a tool called Designview for doing database design using FDs. It lets you check whether a relation is in a particular normal form, test whether decompositions have nice properties, compute attribute closures, try several decomposition sequences and switch between them, generate SQL statements to create the final database schema, and so on.

1. Use Designview to check your answers to exercises that call for computing closures, testing normal forms, decomposing into a desired normal form, and so on.
2. (*Note to instructors: This exercise should be made more specific by providing additional details. See Appendix B.*) Apply Designview to a large, real-world design problem.

BIBLIOGRAPHIC NOTES

Textbook presentations of dependency theory and its use in database design include [3, 38, 436, 443, 656]. Good survey articles on the topic include [663, 355].

FDs were introduced in [156], along with the concept of 3NF, and axioms for inferring FDs were presented in [31]. BCNF was introduced in [157]. The concept of a legal relation instance and dependency satisfaction are studied formally in [279]. FDs were generalized to semantic data models in [674].

Finding a key is shown to be NP-complete in [432]. Lossless-join decompositions were studied in [24, 437, 546]. Dependency-preserving decompositions were studied in [61]. [68] introduced minimal covers. Decomposition into 3NF is studied by [68, 85] and decomposition into BCNF is addressed in [651]. [351] shows that testing whether a relation is in 3NF is NP-complete. [215] introduced 4NF and discussed decomposition into 4NF. Fagin introduced other normal forms in [216] (project-join normal form) and [217] (domain-key normal form). In contrast to the extensive study of vertical decompositions, there has been relatively little formal investigation of horizontal decompositions. [175] investigates horizontal decompositions.

MVDs were discovered independently by Delobel [177], Fagin [215], and Zaniolo [690]. Axioms for FDs and MVDs were presented in [60]. [516] shows that there is no axiomatization for JDs, although [575] provides an axiomatization for a more general class of dependencies. The sufficient conditions for 4NF and 5NF in terms of FDs that were discussed in Section 15.8 are from [171]. An approach to database design that uses dependency information to construct sample relation instances is described in [442, 443].

22

INTERNET DATABASES

He profits most who serves best.

—Motto for Rotary International

The proliferation of computer networks, including the Internet and corporate 'intranets,' has enabled users to access a large number of data sources. This increased access to databases is likely to have a great practical impact; data and services can now be offered directly to customers in ways that were impossible until recently. **Electronic commerce** applications cover a broad spectrum; examples include purchasing books through a Web retailer such as Amazon.com, engaging in online auctions at a site such as eBay, and exchanging bids and specifications for products between companies. The emergence of standards such as XML for describing content (in addition to the presentation aspects) of documents is likely to further accelerate the use of the Web for electronic commerce applications.

While the first generation of Internet sites were collections of HTML files—HTML is a standard for describing how a file should be displayed—most major sites today store a large part (if not all) of their data in database systems. They rely upon DBMSs to provide fast, reliable responses to user requests received over the Internet; this is especially true of sites for electronic commerce. This unprecedented access will lead to increased and novel demands upon DBMS technology. The impact of the Web on DBMSs, however, goes beyond just a new source of large numbers of concurrent queries: The presence of large collections of unstructured text documents and partially structured HTML and XML documents and new kinds of queries such as keyword search challenge DBMSs to significantly expand the data management features they support. In this chapter, we discuss the role of DBMSs in the Internet environment and the new challenges that arise.

We introduce the World Wide Web, Web browsers, Web servers, and the HTML markup language in Section 22.1. In Section 22.2, we discuss alternative architectures for making databases accessible through the Web. We discuss XML, an emerging standard for document description that is likely to supersede HTML, in Section 22.3. Given the proliferation of text documents on the Web, searching them for user-specified keywords is an important new query type. *Boolean* keyword searches ask for documents containing a specified boolean combination of keywords. *Ranked* keyword searches ask for documents that are most relevant to a given list of keywords. We

consider indexing techniques to support boolean keyword searches in Section 22.4 and techniques to support ranked keyword searches in Section 22.5.

22.1 THE WORLD WIDE WEB

The Web makes it possible to access a file anywhere on the Internet. A file is identified by a **universal resource locator (URL)**:

```
http://www.informatik.uni-trier.de/~ley/db/index.html
```

This URL identifies a file called `index.html`, stored in the directory `~ley/db/` on machine `www.informatik.uni-trier.de`. This file is a document formatted using **HyperText Markup Language (HTML)** and contains several **links** to other files (identified through their URLs).

The formatting commands are interpreted by a **Web browser** such as Microsoft's Internet Explorer or Netscape Navigator to display the document in an attractive manner, and the user can then navigate to other related documents by choosing links. A collection of such documents is called a **Web site** and is managed using a program called a **Web server**, which accepts URLs and returns the corresponding documents. Many organizations today maintain a Web site. (Incidentally, the URL shown above is the entry point to Michael Ley's Databases and Logic Programming (DBLP) Web site, which contains information on database and logic programming research publications. It is an invaluable resource for students and researchers in these areas.) The **World Wide Web**, or **Web**, is the collection of Web sites that are accessible over the Internet.

An HTML link contains a URL, which identifies the site containing the linked file. When a user clicks on a link, the Web browser connects to the Web server at the destination Web site using a connection protocol called **HTTP** and submits the link's URL. When the browser receives a file from a Web server, it checks the file type by examining the extension of the file name. It displays the file according to the file's type and if necessary calls an application program to handle the file. For example, a file ending in `.txt` denotes an unformatted text file, which the Web browser displays by interpreting the individual ASCII characters in the file. More sophisticated document structures can be encoded in HTML, which has become a standard way of structuring Web pages for display. As another example, a file ending in `.doc` denotes a Microsoft Word document and the Web browser displays the file by invoking Microsoft Word.

22.1.1 Introduction to HTML

HTML is a simple language used to describe a document. It is also called a **markup language** because HTML works by augmenting regular text with 'marks' that hold special meaning for a Web browser handling the document. Commands in the language

```
<HTML>
<HEAD></HEAD>
<BODY>
Science:
<UL>
  <LI>Author: Richard Feynman</LI>
  <LI>Title:  The Character of Physical Law</LI>
  <LI>Published 1980</LI>
  <LI>Hardcover</LI>
</UL>
Fiction:
<UL>
  <LI>Author: R.K. Narayan</LI>
  <LI>Title:  Waiting for the Mahatma</LI>
  <LI>Published 1981</LI>
<UL>
  <LI>Name:  R.K. Narayan</LI>
  <LI>Title: The English Teacher</LI>
  <LI>Published 1980</LI>
  <LI>Paperback</LI>
</UL>
</BODY>
</HTML>
```

Figure 22.1 Book Listing in HTML

are called **tags** and they consist (usually) of a **start tag** and an **end tag** of the form <TAG> and </TAG>, respectively. For example, consider the HTML fragment shown in Figure 22.1. It describes a Web page that shows a list of books. The document is enclosed by the tags <HTML> and </HTML>, marking it as an HTML document. The remainder of the document—enclosed in <BODY> ... </BODY>—contains information about three books. Data about each book is represented as an unordered list (UL) whose entries are marked with the LI tag. HTML defines the set of valid tags as well as the meaning of the tags. For example, HTML specifies that the tag <TITLE> is a valid tag that denotes the title of the document. As another example, the tag <UL> always denotes an unordered list.

Audio, video, and even programs (written in Java, a highly portable language) can be included in HTML documents. When a user retrieves such a document using a suitable browser, images in the document are displayed, audio and video clips are played, and embedded programs are executed at the user's machine; the result is a rich multimedia presentation. The ease with which HTML documents can be created—

there are now visual editors that automatically generate HTML—and accessed using Internet browsers has fueled the explosive growth of the Web.

22.1.2 Databases and the Web

The Web is the cornerstone of electronic commerce. Many organizations offer products through their Web sites, and customers can place orders by visiting a Web site. For such applications a URL must identify more than just a file, however rich the contents of the file; a URL must provide an entry point to services available on the Web site. It is common for a URL to include a form that users can fill in to describe what they want. If the requested URL identifies a form, the Web server returns the form to the browser, which displays the form to the user. After the user fills in the form, the form is returned to the Web server, and the information filled in by the user can be used as parameters to a program executing at the same site as the Web server.

The use of a Web browser to invoke a program at a remote site leads us to the role of databases on the Web: The invoked program can generate a request to a database system. This capability allows us to easily place a database on a computer network, and make services that rely upon database access available over the Web. This leads to a new and rapidly growing source of concurrent requests to a DBMS, and with thousands of concurrent users routinely accessing popular Web sites, new levels of scalability and robustness are required.

The diversity of information on the Web, its distributed nature, and the new uses that it is being put to lead to challenges for DBMSs that go beyond simply improved performance in traditional functionality. For instance, we require support for queries that are run periodically or continuously and that access data from several distributed sources. As an example, a user may want to be notified whenever a new item meeting some criteria (e.g., a Peace Bear Beanie Baby toy costing less than $15) is offered for sale at one of several Web sites. Given many such user profiles, how can we efficiently monitor them and notify users promptly as the items they are interested in become available? As another instance of a new class of problems, the emergence of the XML standard for describing data leads to challenges in managing and querying XML data (see Section 22.3).

22.2 ARCHITECTURE

To execute a program at the Web server's site, the server creates a new process and communicates with this process using the **common gateway interface (CGI)** protocol. The results of the program can be used to create an HTML document that is returned to the requestor. Pages that are computed in this manner at the time they

```
<HTML><HEAD><TITLE>The Database Bookstore</TITLE></HEAD>
<BODY>
<FORM action="find_books.cgi"  method=post>
   Type an author name:
   <INPUT type="text" name="authorName" size=30 maxlength=50>
   <INPUT type="submit" value="Send it">
   <INPUT type="reset" value="Clear form">
</FORM>
</BODY></HTML>
```

Figure 22.2 A Sample Web Page with Form Input

are requested are called **dynamic pages**; pages that exist and are simply delivered to the Web browser are called **static pages**.

As an example, consider the sample page shown in Figure 22.2. This Web page contains a form where a user can fill in the name of an author. If the user presses the 'Send it' button, the Perl script 'findBooks.cgi' mentioned in Figure 22.2 is executed as a separate process. The CGI protocol defines how the communication between the form and the script is performed. Figure 22.3 illustrates the processes created when using the CGI protocol.

Figure 22.4 shows an example CGI script, written in Perl. We have omitted error-checking code for simplicity. Perl is an interpreted language that is often used for CGI scripting and there are many Perl libraries called **modules** that provide high-level interfaces to the CGI protocol. We use two such libraries in our example: DBI and CGI. DBI is a database-independent API for Perl that allows us to abstract from the DBMS being used—DBI plays the same role for Perl as JDBC does for Java. Here we use DBI as a bridge to an ODBC driver that handles the actual connection to the database. The CGI module is a convenient collection of functions for creating CGI scripts. In part 1 of the sample script, we extract the content of the HTML form as follows:

```
$authorName = $dataIn->param('authorName');
```

Note that the parameter name `authorName` was used in the form in Figure 22.2 to name the first input field. In part 2 we construct the actual SQL command in the variable `$sql`. In part 3 we start to assemble the page that is returned to the Web browser. We want to display the result rows of the query as entries in an unordered list, and we start the list with its start tag `<UL>`. Individual list entries will be enclosed by the `<LI>` tag. Conveniently, the CGI protocol abstracts the actual implementation of how the Web page is returned to the Web browser; the Web page consists simply of the

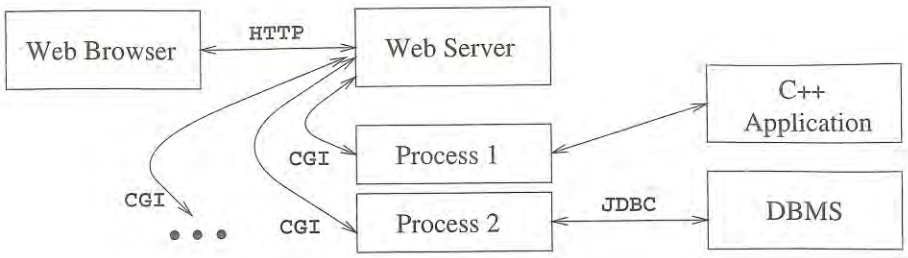


Figure 22.3 Process Structure with CGI Scripts

output of our program. Thus, everything that the script writes in `print`-statements will be part of the dynamically constructed Web page that is returned to the Web browser. Part 4 establishes the database connection and prepares and executes the SQL statement that we stored in the variable `$sql` in part 2. In part 5, we fetch the result of the query, one row at a time, and append each row to the output. Part 6 closes the connection to the database system and we finish in part 7 by appending the closing format tags to the resulting page.

Alternative protocols, in which the program invoked by a request is executed within the Web server process, have been proposed by Microsoft (*Internet Server API (ISAPI)*) and by Netscape (*Netscape Server API (NSAPI)*). Indeed, the TPC-C benchmark has been executed, with good results, by sending requests from 1,500 PC clients to a Web server and through it to an SQL database server.

22.2.1 Application Servers and Server-Side Java

In the previous section, we discussed how the CGI protocol can be used to dynamically assemble Web pages whose content is computed on demand. However, since each page request results in the creation of a new process this solution does not scale well to a large number of simultaneous requests. This performance problem led to the development of specialized programs called **application servers**. An application server has pre-forked threads or processes and thus avoids the startup cost of creating a new process for each request. Application servers have evolved into flexible middle tier packages that provide many different functions in addition to eliminating the process-creation overhead:

- **Integration of heterogeneous data sources:** Most companies have data in many different database systems, from legacy systems to modern object-relational systems. Electronic commerce applications require integrated access to all these data sources.
- **Transactions involving several data sources:** In electronic commerce applications, a user transaction might involve updates at several data sources. An

```
#!/usr/bin/perl
use DBI; use CGI;

### part 1
$dataIn = new CGI;
$dataIn->header();
$authorName = $dataIn->param('authorName');

### part 2
$sql = "SELECT authorName, title FROM books ";
$sql += "WHERE authorName = " + $authorName;

### part 3
print "<HTML><TITLE>Results:</TITLE><UL> Results:";

### part 4
$dbh = DBI->connect( 'DBI:ODBC:BookStore', 'webuser', 'password');
$sth = $dbh->prepare($sql);
$sth->execute;

### part 5
while ( @row = $sth->fetchrow ) {
    print "<LI> @row </LI> \n";
}

### part 6
$sth->finish;
$dbhandle->disconnect;

### part 7
print "</UL></HTML>";
exit;
```

Figure 22.4 A Simple Perl Script

An example of a real application server—IBM WebSphere: IBM WebSphere is an application server that provides a wide range of functionality. It includes a full-fledged Web server and supports dynamic Web page generation. WebSphere includes a Java Servlet run time environment that allows users to extend the functionality of the server. In addition to Java Servlets, Websphere supports other Web technologies such as Java Server Pages and JavaBeans. It includes a connection manager that handles a pool of relational database connections and caches intermediate query results.

application server can ensure transactional semantics across data sources by providing atomicity, isolation, and durability. The **transaction boundary** is the point at which the application server provides transactional semantics. If the transaction boundary is at the application server, very simple client programs are possible.

- **Security:** Since the users of a Web application usually include the general population, database access is performed using a general-purpose user identifier that is known to the application server. While communication between the server and the application at the server side is usually not a security risk, communication between the client (Web browser) and the Web server could be a security hazard. Encryption is usually performed at the Web server, where a secure protocol (in most cases the **Secure Sockets Layer (SSL)** protocol) is used to communicate with the client.

- **Session management:** Often users engage in business processes that take several steps to complete. Users expect the system to maintain continuity during a session, and several session identifiers such as **cookies**, URL extensions, and hidden fields in HTML forms can be used to identify a session. Application servers provide functionality to detect when a session starts and ends and to keep track of the sessions of individual users.

A possible architecture for a Web site with an application server is shown in Figure 22.5. The client (a Web browser) interacts with the Web server through the HTTP protocol. The Web server delivers static HTML or XML pages directly to the client. In order to assemble dynamic pages, the Web server sends a request to the application server. The application server contacts one or more data sources to retrieve necessary data or sends update requests to the data sources. After the interaction with the data sources is completed, the application server assembles the Web page and reports the result to the Web server, which retrieves the page and delivers it to the client.

The execution of business logic at the Web server's site, or **server-side processing**, has become a standard model for implementing more complicated business processes on the Internet. There are many different technologies for server-side processing and

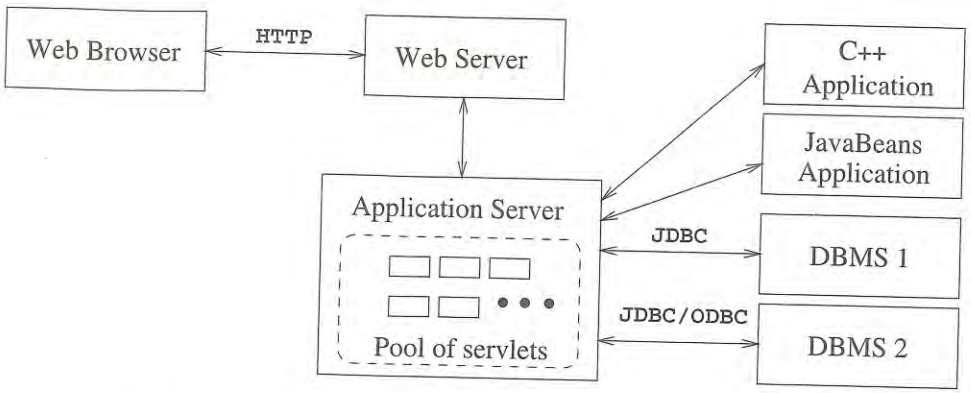


Figure 22.5 Process Structure in the Application Server Architecture

we only mention a few in this section; the interested reader is referred to the references at the end of the chapter.

The **Java Servlet API** allows Web developers to extend the functionality of a Web server by writing small Java programs called **servlets** that interact with the Web server through a well-defined API. A servlet consists of mostly business logic and routines to format relatively small datasets into HTML. Java servlets are executed in their own threads. Servlets can continue to execute even after the client request that led to their invocation is completed and can thus maintain persistent information between requests. The Web server or application server can manage a pool of servlet threads, as illustrated in Figure 22.5, and can therefore avoid the overhead of process creation for each requests. Since servlets are written in Java, they are portable between Web servers and thus allow platform-independent development of server-side applications.

Server-side applications can also be written using JavaBeans. **JavaBeans** are reusable software components written in Java that perform well-defined tasks and can be conveniently packaged and distributed (together with any Java classes, graphics, and other files they need) in **JAR** files. JavaBeans can be assembled to create larger applications and can be easily manipulated using visual tools.

Java Server Pages (JSP) are yet another platform-independent alternative for generating dynamic content on the server side. While servlets are very flexible and powerful, slight modifications, for example in the appearance of the output page, require the developer to change the servlet and to recompile the changes. JSP is designed to separate application logic from the appearance of the Web page, while at the same time simplifying and increasing the speed of the development process. JSP separates content from presentation by using special HTML tags inside a Web page to generate dynamic content. The Web server interprets these tags and replaces them with dynamic content before returning the page to the browser.

For example, consider the following Web page that includes JSP commands:

```
<HTML>
<H1>Hello</H1>
<P>Today is </P>
<jsp:useBean id=="clock" class=="calendar.jspCalendar" />
<UL>
  <LI>Day: <%==clock.getDayOfMonth() %>
  <LI>Year: <%==clock.getYear() %>
</UL>
</HTML>
```

We first load a JavaBean component through the tag `<jsp:useBean>` and then evaluate the `getDayOfMonth` member functions of the bean as marked in the directive `<%==clock.getDayOfMonth() %>`.

The technique of including proprietary markup tags into an HTML file and dynamically evaluating the contents of these tags while assembling the answer page is used in many application servers. Such pages are generally known as **HTML templates**. For example, Cold Fusion is an application server that allows special markup tags that can include database queries. The following code fragment shows an example query:

```
<cfquery name="listBooks" datasource="books">
  select * from books
</cfquery>
```

22.3 BEYOND HTML

While HTML is adequate to represent the structure of documents for display purposes, the features of the language are not sufficient to represent the structure of data within a document for more general applications than a simple display. For example, we can send the HTML document shown in Figure 22.1 to another application and the application can display the information about our books, but using the HTML tags the application cannot distinguish the first names of the authors from their last names. (The application can try to recover such information by looking at the text inside the tags, but that defeats the purpose of structuring the data using tags.) Thus, HTML is inadequate for the exchange of complex documents containing product specifications or bids, for example.

Extensible Markup Language (XML) is a markup language that was developed to remedy the shortcomings of HTML. In contrast to having a fixed set of tags whose meaning is fixed by the language (as in HTML), XML allows the user to define new collections of tags that can then be used to structure any type of data or document the

The design goals of XML: XML was developed starting in 1996 by a working group under guidance of the World Wide Web Consortium (W3C) XML Special Interest Group. The design goals for XML included the following:

1. XML should be compatible with SGML.
2. It should be easy to write programs that process XML documents.
3. The design of XML should be formal and concise.

user wishes to transmit. XML is an important bridge between the document-oriented view of data implicit in HTML and the schema-oriented view of data that is central to a DBMS. It has the potential to make database systems more tightly integrated into Web applications than ever before.

XML emerged from the confluence of two technologies, SGML and HTML. The **Standard Generalized Markup Language (SGML)** is a metalanguage that allows the definition of data and document interchange languages such as HTML. The SGML standard was published in 1988 and many organizations that manage a large number of complex documents have adopted it. Due to its generality, SGML is complex and requires sophisticated programs to harness its full potential. XML was developed to have much of the power of SGML while remaining relatively simple. Nonetheless, XML, like SGML, allows the definition of new document markup languages.

Although XML does not prevent a user from designing tags that encode the display of the data in a Web browser, there is a style language for XML called **Extensible Style Language (XSL)**. XSL is a standard way of describing how an XML document that adheres to a certain vocabulary of tags should be displayed.

22.3.1 Introduction to XML

The short introduction to XML given in this section is not complete, and the references at the end of this chapter provide starting points for the interested reader. We will use the small XML document shown in Figure 22.6 as an example.

- **Elements**. Elements, also called tags, are the primary building blocks of an XML document. The start of the content of an element `ELM` is marked with `<ELM>`, which is called the **start tag**, and the end of the content end is marked with `</ELM>`, called the **end tag**. In our example document, the element `BOOKLIST` encloses all information in the sample document. The element `BOOK` demarcates all data associated with a single book. XML elements are case sensitive: the element `<BOOK>` is different from `<Book>`. Elements must be properly nested. Start tags

that appear inside the content of other tags must have a corresponding end tag. For example, consider the following XML fragment:

```
<BOOK>
  <AUTHOR>
    <FIRSTNAME>Richard</FIRSTNAME><LASTNAME>Feynman</LASTNAME>
  </AUTHOR>
</BOOK>
```

The element `AUTHOR` is completely nested inside the element `BOOK`, and both the elements `LASTNAME` and `FIRSTNAME` are nested inside the element `AUTHOR`.

- **Attributes.** An element can have descriptive attributes that provide additional information about the element. The values of attributes are set inside the start tag of an element. For example, let `ELM` denote an element with the attribute `att`. We can set the value of `att` to `value` through the following expression: `<ELM att="value">`. All attribute values must be enclosed in quotes. In Figure 22.6, the element `BOOK` has two attributes. The attribute `genre` indicates the genre of the book (science or fiction) and the attribute `format` indicates whether the book is a hardcover or a paperback.

- **Entity references.** Entities are shortcuts for portions of common text or the content of external files and we call the usage of an entity in the XML document an **entity reference**. Wherever an entity reference appears in the document, it is textually replaced by its content. Entity references start with a '`&`' and end with a '`;`' . There are five predefined entities in XML that are placeholders for characters with special meaning in XML. For example, the < character that marks the beginning of an XML command is reserved and has to be represented by the entity `lt`. The other four reserved characters are &, >, ; and ' and they are represented by the entities `amp`, `gt`, `quot`, and `apos`. For example, the text '1 < 5' has to be encoded in an XML document as follows: `"1<5"`. We can also use entities to insert arbitrary Unicode characters into the text. **Unicode** is a standard for character representations, and is similar to ASCII. For example, we can display the Japanese Hiragana character 'a' using the entity reference `&#x3042`.

- **Comments.** We can insert comments anywhere in an XML document. Comments start with `<!-` and end with `->`. Comments can contain arbitrary text except the string `--`.

- **Document type declarations (DTDs).** In XML, we can define our own markup language. A DTD is a set of rules that allows us to specify our own set of elements, attributes, and entities. Thus, a DTD is basically a grammar that indicates what tags are allowed, in what order they can appear, and how they can be nested. We will discuss DTDs in detail in the next section.

We call an XML document **well-formed** if it does not have an associated DTD but follows the following structural guidelines:

```
<?XML version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE BOOKLIST SYSTEM "emp.dtd">
<BOOKLIST>
<BOOK genre="Science" format="Hardcover">
  <AUTHOR>
    <FIRSTNAME>Richard</FIRSTNAME><LASTNAME>Feynman</LASTNAME>
  </AUTHOR>
  <TITLE>The Character of Physical Law</TITLE>
  <PUBLISHED>1980</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
  <AUTHOR>
    <FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
  </AUTHOR>
  <TITLE>Waiting for the Mahatma</TITLE>
  <PUBLISHED>1981</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
  <AUTHOR>
    <FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
  </AUTHOR>
  <TITLE>The English Teacher</TITLE>
  <PUBLISHED>1980</PUBLISHED>
</BOOK>
</BOOKLIST>
```

Figure 22.6 Book Information in XML

- The document starts with an XML declaration. An example of an XML declaration is the first line of the XML document shown in Figure 22.6.
- There is a root element that contains all the other elements. In our example, the root element is the element `BOOKLIST`.
- All elements must be properly nested. This requirement states that start and end tags of an element must appear within the same enclosing element.

22.3.2 XML DTDs

A DTD is a set of rules that allows us to specify our own set of elements, attributes, and entities. A DTD specifies which elements we can use and constraints on these elements, e.g., how elements can be nested and where elements can appear in the

```
<!DOCTYPE BOOKLIST [
  <!ELEMENT BOOKLIST (BOOK)*>
  <!ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?)>
    <!ELEMENT AUTHOR (FIRSTNAME,LASTNAME)>
      <!ELEMENT FIRSTNAME (#PCDATA)>
      <!ELEMENT LASTNAME (#PCDATA)>
    <!ELEMENT TITLE (#PCDATA)>
    <!ELEMENT PUBLISHED (#PCDATA)>
  <!ATTLIST BOOK genre (Science|Fiction) #REQUIRED>
  <!ATTLIST BOOK format (Paperback|Hardcover) "Paperback">
]>
```

Figure 22.7 Bookstore XML DTD

document. We will call a document **valid** if there is a DTD associated with it and the document is structured according to the rules set by the DTD. In the remainder of this section, we will use the example DTD shown in Figure 22.7 to illustrate how to construct DTDs.

A DTD is enclosed in `<!DOCTYPE name [DTDdeclaration]>`, where `name` is the name of the outermost enclosing tag, and DTDdeclaration is the text of the rules of the DTD. The DTD starts with the outermost element, also called the root element, which is `BOOKLIST` in our example. Consider the next rule:

```
<!ELEMENT BOOKLIST (BOOK)*>
```

This rule tells us that the element `BOOKLIST` consists of zero or more `BOOK` elements. The * after `BOOK` indicates how many `BOOK` elements can appear inside the `BOOKLIST` element. A * denotes zero or more occurrences, a + denotes one or more occurrences, and a ? denotes zero or one occurrence. For example, if we want to ensure that a `BOOKLIST` has at least one book, we could change the rule as follows:

```
<!ELEMENT BOOKLIST (BOOK)+>
```

Let us look at the next rule:

```
<!ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?)>
```

This rule states that a `BOOK` element contains a `NAME` element, a `TITLE` element, and an optional `PUBLISHED` element. Note the use of the ? to indicate that the information is optional by having zero or one occurrence of the element. Let us move ahead to the following rule:

```
<!ELEMENT LASTNAME (#PCDATA)>
```

Until now we only considered elements that contained other elements. The above rule states that LASTNAME is an element that does not contain other elements, but contains actual text. Elements that only contain other elements are said to have **element content**, whereas elements that also contain #PCDATA are said to have **mixed content**. In general, an element type declaration has the following structure:

```
<!ELEMENT (contentType) >
```

The are five possible content types:

- Other elements.
- The special symbol #PCDATA, which indicates (parsed) character data.
- The special symbol EMPTY, which indicates that the element has no content. Elements that have no content are not required to have an end tag.
- The special symbol ANY, which indicates that any content is permitted. This content should be avoided whenever possible since it disables all checking of the document structure inside the element.
- A **regular expression** constructed from the above four choices. A regular expression is one of the following:
  - exp1, exp2, exp3: A list of regular expressions.
  - exp*: An optional expression (zero or more occurrences).
  - exp?: An optional expression (zero or one occurrences).
  - exp+: A mandatory expression (one or more occurrences).
  - exp1 | exp2: exp1 or exp2.

Attributes of elements are declared outside of the element. For example, consider the following attribute declaration from Figure 22.7.

```
<!ATTLIST BOOK genre (Science|Fiction) #REQUIRED>
```

This XML DTD fragment specifies the attribute genre, which is an attribute of the element BOOK. The attribute can take two values: Science or Fiction. Each BOOK element must be described in its start tag by a genre attribute since the attribute is required as indicated by #REQUIRED. Let us look at the general structure of a DTD attribute declaration:

```
<!ATTLIST elementName (attName attType default)+ >
```

The keyword `ATTLIST` indicates the beginning of an attribute declaration. The string `elementName` is the name of the element that the following attribute definition is associated with. What follows is the declaration of one or more attributes. Each attribute has a name as indicated by `attName` and a type as indicated by `attType`. XML defines several possible types for an attribute. We only discuss **string types** and **enumerated types** here. An attribute of type string can take any string as a value. We can declare such an attribute by setting its type field to `CDATA`. For example, we can declare a third attribute of type string of the element `BOOK` as follows:

```
<!ATTLIST BOOK edition CDATA "1">
```

If an attribute has an enumerated type, we list all its possible values in the attribute declaration. In our example, the attribute `genre` is an enumerated attribute type; its possible attribute values are 'Science' and 'Fiction'.

The last part of an attribute declaration is called its **default specification**. The XML DTD in Figure 22.7 shows two different default specifications: `#REQUIRED` and the string 'Paperback'. The default specification `#REQUIRED` indicates that the attribute is required and whenever its associated element appears somewhere in the XML document a value for the attribute must be specified. The default specification indicated by the string 'Paperback' indicates that the attribute is not required; whenever its associated element appears without setting a value for the attribute, the attribute automatically takes the value 'Paperback'. For example, we can make the attribute value 'Science' the default value for the `genre` attribute as follows:

```
<!ATTLIST BOOK genre (Science|Fiction) "Science">
```

The complete XML DTD language is much more powerful than the small part that we have explained. The interested reader is referred to the references at the end of the chapter.

22.3.3 Domain-Specific DTDs

Recently, DTDs have been developed for several specialized domains—including a wide range of commercial, engineering, financial, industrial, and scientific domains—and a lot of the excitement about XML has its origins in the belief that more and more standardized DTDs will be developed. Standardized DTDs would enable seamless data exchange between heterogeneous sources, a problem that is solved today either by implementing specialized protocols such as **Electronic Data Interchange (EDI)** or by implementing ad hoc solutions.

Even in an environment where all XML data is valid, it is not possible to straightforwardly integrate several XML documents by matching elements in their DTDs because

even when two elements have identical names in two different DTDs, the meaning of the elements could be completely different. If both documents use a single, standard DTD we avoid this problem. The development of standardized DTDs is more a social process than a hard research problem since the major players in a given domain or industry segment have to collaborate.

For example, the **mathematical markup language (MathML)** has been developed for encoding mathematical material on the Web. There are two types of MathML elements. The 28 **presentation elements** describe the layout structure of a document; examples are the `mrow` element, which indicates a horizontal row of characters, and the `msup` element, which indicates a base and a subscript. The 75 **content elements** describe mathematical concepts. An example is the `plus` element which denotes the addition operator. (There is a third type of element, the `math` element, that is used to pass parameters to the MathML processor.) MathML allows us to encode mathematical objects in both notations since the requirements of the user of the objects might be different. Content elements encode the precise mathematical meaning of an object without ambiguity and thus the description can be used by applications such as computer algebra systems. On the other hand, good notation can suggest the logical structure to a human and can emphasize key aspects of an object; presentation elements allow us to describe mathematical objects at this level.

For example, consider the following simple equation:

$$x^2 - 4x - 32 = 0$$

Using presentation elements, the equation is represented as follows:

```
<mrow>
  <mrow> <msup><mi>x</mi><mn>2</mn></msup>
         <mo>-</mo>
         <mrow><mn>4</mn><mo>&invisibletimes;</mo><mi>x</mi></mrow>
         <mo>-</mo><mn>32</mn>
  </mrow><mo>=</mo><mn>0</mn>
</mrow>
```

Using content elements, the equation is described as follows:

```
<reln><eq/>
  <apply>
    <minus/>
    <apply> <power/> <ci>x</ci> <cn>2</cn> </apply>
    <apply> <times/> <cn>4</cn> <ci>x</ci> </apply>
    <cn>32</cn>
  </apply> <cn>0</cn>
</reln>
```

Note the additional power that we gain from using MathML instead of encoding the formula in HTML. The common way of displaying mathematical objects inside an HTML object is to include images that display the objects, for example as in the following code fragment:

```
<IMG SRC="images/equation.gif" ALT=" x2̂ - 4x - 32 = 10 " >
```

The equation is encoded inside an `IMG` tag with an alternative display format specified in the `ALT` tag. Using this encoding of a mathematical object leads to the following presentation problems. First, the image is usually sized to match a certain font size and on systems with other font sizes the image is either too small or too large. Second, on systems with a different background color the picture does not blend into the background and the resolution of the image is usually inferior when printing the document. Apart from problems with changing presentations, we cannot easily search for a formula or formula fragments on a page, since there is no specific markup tag.

22.3.4 XML-QL: Querying XML Data

Given that data is encoded in a way that reflects (a considerable amount of) structure in XML documents, we have the opportunity to use a high-level language that exploits this structure to conveniently retrieve data from within such documents. Such a language would bring XML data management much closer to database management than the text-oriented paradigm of HTML documents. Such a language would also allow us to easily translate XML data between different DTDs, as is required for integrating data from multiple sources.

At the time of writing of this chapter (summer of 1999), the discussion about a standard query language for XML was still ongoing. In this section, we will give an informal example of one specific query language for XML called **XML-QL** that has strong similarities to several query languages that have been developed in the database community (see Section 22.3.5).

Consider again the XML document shown in Figure 22.6. The following example query returns the last names of all authors, assuming that our XML document resides at the location `www.ourbookstore.com/books.xml`.

```
WHERE <BOOK>
        <NAME><LASTNAME> $1 </LASTNAME></NAME>
      </BOOK> IN "www.ourbookstore.com/books.xml"
CONSTRUCT <RESULTNAME> $1 </RESULTNAME>
```

This query extracts data from the XML document by specifying a pattern of markups. We are interested in data that is nested inside a `BOOK` element, a `NAME` element, and

a LASTNAME element. For each part of the XML document that matches the structure specified by the query, the variable l is bound to the contents of the element LASTNAME. To distinguish variable names from normal text, variable names are prefixed with a dollar sign $. If this query is applied to the sample data shown in Figure 22.6, the result would be the following XML document:

```
<RESULTNAME>Feynman</RESULTNAME>
<RESULTNAME>Narayan</RESULTNAME>
```

Selections are expressed by placing text in the content of an element. Also, the output of a query is not limited to a single element. We illustrate these two points in the next query. Assume that we want to find the last names and first names of all authors who wrote a book that was published in 1980. We can express this query as follows:

```
WHERE <BOOK> <NAME>
          <LASTNAME> $l </LASTNAME>
          <FIRSTNAME> $f </FIRSTNAME>
        </NAME>
        <PUBLISHED>1980</PUBLISHED>
      </BOOK> IN "www.ourbookstore.com/books.xml"
CONSTRUCT    <RESULTNAME>
               <FIRST>$f</FIRST><LAST>$l</LAST>
             </RESULTNAME>
```

The result of the above query is the following XML document:

```
<RESULTNAME><FIRST>Richard</FIRST><LAST>Feynman</LAST></RESULTNAME>
<RESULTNAME><FIRST>R.K.</FIRST><LAST>Narayan</LAST></RESULTNAME>
```

We conclude our discussion with a slightly more complicated example. Suppose that for each year we want to find the last names of authors who wrote a book published in that year. We group by PUBLISHED and the result contains a list of last names for each year:

```
WHERE <BOOK> $e <BOOK> IN "www.ourbookstore.com/books.xml",
      <AUTHOR>$n</AUTHOR>,
      <PUBLISHED>$p</PUBLISHED> IN $e
CONSTRUCT <RESULT><PUBLISHED> $p </PUBLISHED>
      WHERE <LASTNAME> $l </LASTNAME> IN $n
      CONSTRUCT <LASTNAME> $l </LASTNAME>
    </RESULT>
```

Using the XML document in Figure 22.6 as input, this query produces the following result:

Commercial database systems and XML: Many relational and object-relational database system vendors are currently looking into support for XML in their database engines. Several vendors of object-oriented database management systems already offer database engines that can store XML data whose contents can be accessed through graphical user interfaces, server-side Java extensions, or through XML-QL queries.

```
<RESULT> <PUBLISHED>1980</PUBLISHED>
  <LASTNAME>Feynman</LASTNAME>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
<RESULT> <PUBLISHED>1981</PUBLISHED>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
```

22.3.5 The Semistructured Data Model

Consider a set of documents on the Web that contain hyperlinks to other documents. These documents, although not completely unstructured, cannot be modeled naturally in the relational data model because the pattern of hyperlinks is not regular across documents. A bibliography file also has a certain degree of structure due to fields such as *author* and *title*, but is otherwise unstructured text. While some data is completely unstructured—for example video streams, audio streams, and image data—a lot of data is neither completely unstructured nor completely structured. We refer to data with partial structure as **semistructured data**. XML documents represent an important and growing source of semistructured data, and the theory of semistructured data models and queries has the potential to serve as the foundation for XML.

There are many reasons why data might be semistructured. First, the structure of data might be implicit, hidden, unknown, or the user might choose to ignore it. Second, consider the problem of integrating data from several heterogeneous sources where data exchange and transformation are important problems. We need a highly flexible data model to integrate data from all types of data sources including flat files and legacy systems; a structured data model is often too rigid. Third, we cannot query a structured database without knowing the schema, but sometimes we want to query the data without full knowledge of the schema. For example, we cannot express the query "Where in the database can we find the string Malgudi?" in a relational database system without knowing the schema.

All data models proposed for semistructured data represent the data as some kind of labeled graph. Nodes in the graph correspond to compound objects or atomic values,

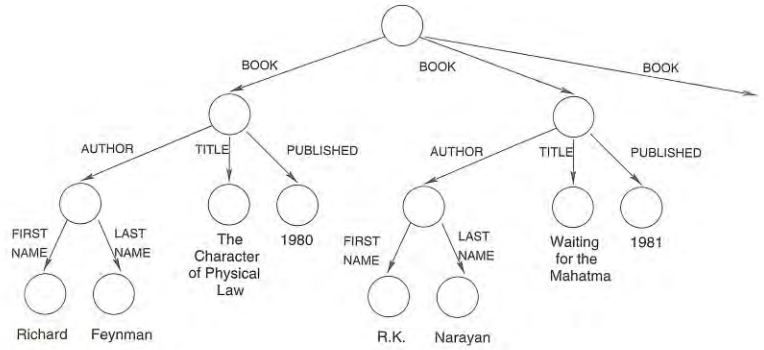


Figure 22.8 The Semistructured Data Model

and edges correspond to attributes. There is no separate schema and no auxiliary description; the data in the graph is self describing. For example, consider the graph shown in Figure 22.8, which represents part of the XML data from Figure 22.6. The root node of the graph represents the outermost element, `BOOKLIST`. The node has three outgoing edges that are labeled with the element name `BOOK`, since the list of books consists of three individual books.

We now discuss one of the proposed data models for semistructured data, called the **object exchange model (OEM)**. Each object is described by a triple consisting of a *label,* a *type,* and the *value* of the object. (An object in the object exchange model also has an object identifier, which is a unique identifier for the object. We omit object identifiers from our discussion here; the references at the end of the chapter provide pointers for the interested reader.) Since each object has a label that can be thought of as the column name in the relational model, and each object has a type that can be thought of as the column type in the relational model, the object exchange model is basically self-describing. Labels in the object exchange model should be as informative as possible, since they can serve two purposes: They can be used to identify an object as well as to convey the meaning of an object. For example, we can represent the last name of an author as follows:

⟨`lastName, string, "Feynman"`⟩

More complex objects are decomposed hierarchically into smaller objects. For example, an author name can contain a first name and a last name. This object is described as follows:

⟨`authorName, set,` $\{firstname_1, lastname_1\}$⟩
  $firstname_1$ `is` ⟨`firstName, string, "Richard"`⟩
  $lastname_1$ `is` ⟨`lastName, string, "Feynman"`⟩

As another example, an object representing a set of books is described as follows:

⟨bookList, set, $\{book_1, book_2, book_3\}$⟩
  $book_1$ is ⟨book, set, $\{author_1, title_1, published_1\}$⟩
  $book_2$ is ⟨book, set, $\{author_2, title_2, published_2\}$⟩
  $book_3$ is ⟨book, set, $\{author_3, title_3, published_3\}$⟩
    $author_3$ is ⟨author, set, $\{firstname_3, lastname_3\}$⟩
    $title_3$ is ⟨title, string, "The English Teacher"⟩
    $published_3$ is ⟨published, integer, 1980⟩

22.3.6 Implementation Issues for Semistructured Data

Database system support for semistructured data has been the focus of much research recently, and given the commercial success of XML, this emphasis is likely to continue. Semistructured data poses new challenges, since most of the traditional storage, indexing, and query processing strategies assume that the data adheres to a regular schema. For example, should we store semistructured data by mapping it into the relational model and then store the mapped data in a relational database system? Or does a storage subsystem specialized for semistructured data deliver better performance? How can we index semistructured data? Given a query language like XML-QL, what are suitable query processing strategies? Current research tries to find answers to these questions.

22.4 INDEXING FOR TEXT SEARCH

In this section, we assume that our database is a collection of documents and we call such a database a **text database**. For simplicity, we assume that the database contains exactly one relation and that the relation schema has exactly one field of type document. Thus, each record in the relation contains exactly one document. In practice, the relation schema would contain other fields such as the date of the creation of the document, a possible classification of the document, or a field with keywords describing the document. Text databases are used to store newspaper articles, legal papers, and other types of documents.

An important class of queries based on **keyword search** enables us to ask for all documents containing a given keyword. This is the most common kind of query on the Web today, and is supported by a number of search engines such as AltaVista and Lycos. Some systems maintain a list of synonyms for important words and return documents that contain a desired keyword or one of its synonyms; for example, a query asking for documents containing *car* will also retrieve documents containing *automobile*. A more complex query is "Find all documents that have *keyword1* AND *keyword2*." For such composite queries, constructed with AND, OR, and NOT, we can *rank* retrieved documents by the proximity of the query keywords in the document.

There are two common types of queries for text databases: boolean queries and ranked queries. In a **boolean query**, the user supplies a boolean expression of the following form, which is called **conjunctive normal form**:

$$(t_{11} \vee t_{12} \vee \ldots \vee t_{1i_1}) \wedge \ldots \wedge (t_{j1} \vee t_{12} \vee \ldots \vee t_{1i_j}),$$

where the t_{ij} are individual query terms or keywords. The query consists of j **conjuncts**, each of which consists of several **disjuncts**. In our query, the first conjunct is the expression $(t_{11} \vee t_{12} \vee \ldots \vee t_{1i_1})$; it consists of i_1 disjuncts. Queries in conjunctive normal form have a natural interpretation. The result of the query are documents that involve several concepts. Each conjunct corresponds to one concept, and the different words within each conjunct correspond to different terms for the same concept.

Ranked queries are structurally very similar. In a **ranked query** the user also specifies a list of words, but the result of the query is a list of documents ranked by their relevance to the list of user terms. How to define when and how relevant a document is to a set of user terms is a difficult problem. Algorithms to evaluate such queries belong to the field of **information retrieval**, which is closely related to database management. Information retrieval systems, like database systems, have the goal of enabling users to query a large volume of data, but the focus has been on large collections of unstructured documents. Updates, concurrency control, and recovery have traditionally not been addressed in information retrieval systems because the data in typical applications is largely static.

The criteria used to evaluate such information retrieval systems include **precision**, which is the percentage of retrieved documents that are relevant to the query, and **recall**, which is the percentage of relevant documents in the database that are retrieved in response to a query.

The advent of the Web has given fresh impetus to information retrieval because millions of documents are now available to users and searching for desired information is a fundamental operation; without good search mechanisms, users would be overwhelmed. An index for an information retrieval system essentially contains $\langle keyword, documentid \rangle$ pairs, possibly with additional fields such as the number of times a keyword appears in a document; a Web search engine creates a centralized index for documents that are stored at several sites.

In the rest of this section, we concentrate on boolean queries. We introduce two index schemas that support the evaluation of boolean queries efficiently. The *inverted file* index discussed in Section 22.4.1 is widely used due to its simplicity and good performance. Its main disadvantage is that it imposes a significant space overhead: The size can be up to 300 percent the size of the original file. The *signature file* index discussed in Section 22.4.2 has a small space overhead and offers a quick filter that eliminates most nonqualifying documents. However, it scales less well to larger

Rid	Document	Signature
1	agent James Bond	1100
2	agent mobile computer	1101
3	James Madison movie	1011
4	James Bond movie	1110

Word	Inverted list	Hash
agent	$\langle 1, 2 \rangle$	1000
Bond	$\langle 1, 4 \rangle$	0100
computer	$\langle 2 \rangle$	0100
James	$\langle 1, 3, 4 \rangle$	1000
Madison	$\langle 3 \rangle$	0001
mobile	$\langle 2 \rangle$	0001
movie	$\langle 3, 4 \rangle$	0010

Figure 22.9 A Text Database with Four Records and Indexes

database sizes because the index has to be sequentially scanned. We discuss evaluation of ranked queries in Section 22.5.

We assume that slightly different words that have the same root have been **stemmed**, or analyzed for the common root, during index creation. For example, we assume that the result of a query on 'index' also contains documents that include the terms 'indexes' and 'indexing.' Whether and how to stem is application dependent, and we will not discuss the details.

As a running example, we assume that we have the four documents shown in Figure 22.9. For simplicity, we assume that the record identifiers of the four documents are the numbers one to four. Usually the record identifiers are not physical addresses on the disk, but rather entries in an **address table**. An address table is an array that maps the logical record identifiers, as shown in Figure 22.9, to physical record addresses on disk.

22.4.1 Inverted Files

An **inverted file** is an index structure that enables fast retrieval of all documents that contain a query term. For each term, the index maintains an ordered list (called the **inverted list**) of document identifiers that contain the indexed term. For example, consider the text database shown in Figure 22.9. The query term 'James' has the inverted list of record identifiers $\langle 1, 3, 4 \rangle$ and the query term 'movie' has the list $\langle 3, 4 \rangle$. Figure 22.9 shows the inverted lists of all query terms.

In order to quickly find the inverted list for a query term, all possible query terms are organized in a second index structure such as a B+ tree or a hash index. To avoid any confusion, we will call the second index that allows fast retrieval of the inverted list for a query term the **vocabulary index**. The vocabulary index contains each possible query term and a pointer to its inverted list.

A query containing a single term is evaluated by first traversing the vocabulary index to the leaf node entry with the address of the inverted list for the term. Then the inverted list is retrieved, the rids are mapped to physical document addresses, and the corresponding documents are retrieved. A query with a conjunction of several terms is evaluated by retrieving the inverted lists of the query terms one at a time and intersecting them. In order to minimize memory usage, the inverted lists should be retrieved in order of increasing length. A query with a disjunction of several terms is evaluated by merging all relevant inverted lists.

Consider again the example text database shown in Figure 22.9. To evaluate the query 'James', we probe the vocabulary index to find the inverted list for 'James', fetch the inverted list from disk and then retrieve document one. To evaluate the query 'James' `AND` 'Bond', we retrieve the inverted list for the term 'Bond' and intersect it with the inverted list for the term 'James.' (The inverted list of the term 'Bond' has length two, whereas the inverted list of the term 'James' has length three.) The result of the intersection of the list $\langle 1, 4 \rangle$ with the list $\langle 1, 3, 4 \rangle$ is the list $\langle 1, 4 \rangle$ and the first and fourth document are retrieved. To evaluate the query 'James' `OR` 'Bond,' we retrieve the two inverted lists in any order and merge the results.

22.4.2 Signature Files

A **signature file** is another index structure for text database systems that supports efficient evaluation of boolean queries. A signature file contains an index record for each document in the database. This index record is called the **signature** of the document. Each signature has a fixed size of b bits; b is called the **signature width**. How do we decide which bits to set for a document? The bits that are set depend on the words that appear in the document. We map words to bits by applying a hash function to each word in the document and we set the bits that appear in the result of the hash function. Note that unless we have a bit for each possible word in the vocabulary, the same bit could be set twice by different words because the hash function maps both words to the same bit. We say that a signature S_1 matches another signature S_2 if all the bits that are set in signature S_2 are also set in signature S_1. If signature S_1 matches signature S_2, then signature S_1 has at least as many bits set as signature S_2.

For a query consisting of a conjunction of terms, we first generate the query signature by applying the hash function to each word in the query. We then scan the signature file and retrieve all documents whose signatures match the query signature, because every such document is a potential result to the query. Since the signature does not uniquely identify the words that a document contains, we have to retrieve each potential match and check whether the document actually contains the query terms. A document whose signature matches the query signature but that does not contain all terms in the query is called a **false positive**. A false positive is an expensive mistake since the

breadth. Thus, querying effectively using a boolean keyword search requires expert users who can carefully combine terms specifying a very narrowly defined subject.

One natural solution to the excessive number of answers returned by boolean keyword searches is to take the output of the boolean text query and somehow process this set further to find the most relevant pages. For abstract concepts, however, often the most relevant pages do not contain the search terms at all and are therefore not returned by a boolean keyword search! For example, consider the query term 'Web browser.' A boolean text query using the terms does not return the relevant pages of Netscape Corporation or Microsoft, because these pages do not contain the term 'Web browser' at all. Similarly, the home page of Yahoo does not contain the term 'search engine.' The problem is that relevant sites do not necessarily describe their contents in a way that is useful for boolean text queries.

Until now, we only considered information within a single Web page to estimate its relevance to a query. But Web pages are connected through hyperlinks, and it is quite likely that there is a Web page containing the term 'search engine' that has a link to Yahoo's home page. Can we use the information hidden in such links?

In our search for relevant pages, we distinguish between two types of pages: *authorities* and *hubs*. An **authority** is a page that is very relevant to a certain topic and that is recognized by other pages as authoritative on the subject. These other pages, called hubs, usually have a significant number of hyperlinks to authorities, although they themselves are not very well known and do not necessarily carry a lot of content relevant to the given query. **Hub** pages could be compilations of resources about a topic on a site for professionals, lists of recommended sites for the hobbies of an individual user, or even a part of the bookmarks of an individual user that are relevant to one of the user's interests; their main property is that they have many outgoing links to relevant pages. Good hub pages are often not well known and there may be few links pointing to a good hub. In contrast, good authorities are 'endorsed' by many good hubs and thus have many links from good hub pages.

We will use this symbiotic relationship between hubs and authorities in the HITS algorithm, a link-based search algorithm that discovers high-quality pages that are relevant to a user's query terms.

22.5.1 An Algorithm for Ranking Web Pages

In this section we will discuss HITS, an algorithm that finds good authorities and hubs and returns them as the result of a user query. We view the World Wide Web as a directed graph. Each Web page represents a node in the graph, and a hyperlink from page A to page B is represented as an edge between the two corresponding nodes.

Assume that we are given a user query with several terms. The algorithm proceeds in two steps. In the first step, the *sampling step*, we collect a set of pages called the **base set**. The base set most likely includes very relevant pages to the user's query, but the base set can still be quite large. In the second step, the *iteration step*, we find good authorities and good hubs among the pages in the base set.

The sampling step retrieves a set of Web pages that contain the query terms, using some traditional technique. For example, we can evaluate the query as a boolean keyword search and retrieve all Web pages that contain the query terms. We call the resulting set of pages the **root set**. The root set might not contain all relevant pages because some authoritative pages might not include the user query words. But we expect that at least some of the pages in the root set contain hyperlinks to the most relevant authoritative pages or that some authoritative pages link to pages in the root set. This motivates our notion of a **link page**. We call a page a link page if it has a hyperlink to some page in the root set or if a page in the root set has a hyperlink to it. In order not to miss potentially relevant pages, we augment the root set by all link pages and we call the resulting set of pages the **base set**. Thus, the base set includes all root pages and all link pages; we will refer to a Web page in the base set as a **base page**.

Our goal in the second step of the algorithm is to find out which base pages are good hubs and good authorities and to return the best authorities and hubs as the answers to the query. To quantify the quality of a base page as a hub and as an authority, we associate with each base page in the base set a **hub weight** and an **authority weight**. The hub weight of the page indicates the quality of the page as a hub, and the authority weight of the page indicates the quality of the page as an authority. We compute the weights of each page according to the intuition that a page is a good authority if many good hubs have hyperlinks to it, and that a page is a good hub if it has many outgoing hyperlinks to good authorities. Since we do not have any a priori knowledge about which pages are good hubs and authorities, we initialize all weights to one. We then update the authority and hub weights of base pages iteratively as described below.

Consider a base page p with hub weight h_p and with authority weight a_p. In one iteration, we update a_p to be the sum of the hub weights of all pages that have a hyperlink to p. Formally:

$$a_p = \sum_{\text{All base pages } q \text{ that have a link to } p} h_q$$

Analogously, we update h_p to be the sum of the weights of all pages that p points to:

$$h_p = \sum_{\text{All base pages } q \text{ such that } p \text{ has a link to } q} a_q$$

Computing hub and authority weights: We can use matrix notation to write the updates for all hub and authority weights in one step. Assume that we number all pages in the base set $\{1, 2, ..., n\}$. The adjacency matrix B of the base set is an $n \times n$ matrix whose entries are either 0 or 1. The matrix entry (i, j) is set to 1 if page i has a hyperlink to page j; it is set to 0 otherwise. We can also write the hub weights h and authority weights a in vector notation: $h = \langle h_1, \ldots, h_n \rangle$ and $a = \langle a_1, \ldots, a_n \rangle$. We can now rewrite our update rules as follows:

$$h = B \cdot a, \quad \text{and} \quad a = B^T \cdot h \ .$$

Unfolding this equation once, corresponding to the first iteration, we obtain:

$$h = BB^T h = (BB^T)h, \quad \text{and} \quad a = B^T Ba = (B^T B)a \ .$$

After the second iteration, we arrive at:

$$h = (BB^T)^2 h, \quad \text{and} \quad a = (B^T B)^2 a \ .$$

Results from linear algebra tell us that the sequence of iterations for the hub (resp. authority) weights converges to the principal eigenvectors of BB^T (resp. $B^T B$) if we normalize the weights before each iteration so that the sum of the squares of all weights is always $2 \cdot n$. Furthermore, results from linear algebra tell us that this convergence is independent of the choice of initial weights, as long as the initial weights are positive. Thus, our rather arbitrary choice of initial weights—we initialized all hub and authority weights to 1—does not change the outcome of the algorithm.

Comparing the algorithm with the other approaches to querying text that we discussed in this chapter, we note that the iteration step of the HITS algorithm—the distribution of the weights—does not take into account the words on the base pages. In the iteration step, we are only concerned about the relationship between the base pages as represented by hyperlinks.

The HITS algorithm often produces very good results. For example, the five highest ranked authorities for the query 'search engines' are the following Web pages:

```
http://www.yahoo.com/
http://www.excite.com/
http://www.mckinley.com/
http://www.lycos.com/
http://www.altavista.digital.com/
```

The three highest ranked authorities for the query containing the single keyword 'Gates' are the following Web pages:

```
http://www.roadahead.com/
http://www.microsoft.com/
http://www.microsoft.com/corpinfo/bill-g.htm
```

22.6 POINTS TO REVIEW

- Files on the World Wide Web are identified through *universal resource locators (URLs)*. A *Web browser* takes a URL, goes to the site containing the file, and asks the *Web server* at that site for the file. It then displays the file appropriately, taking into account the type of file and the formatting instructions that it contains. The browser calls application programs to handle certain types of files, e.g., it calls Microsoft Word to handle Word documents (which are identified through a `.doc` file name extension). *HTML* is a simple *markup language* used to describe a document. Audio, video, and even Java programs can be included in HTML documents.

  Increasingly, data accessed through the Web is stored in DBMSs. A Web server can access data in a DBMS to construct a page requested by a Web browser. **(Section 22.1)**

- A Web server often has to execute a program at its site in order to satisfy a request from a Web browser (which is usually executing at a different site). For example, it may have to access data in a DBMS. There are two ways for a Web server to execute a program: It can create a new process and communicate with it using the *CGI* protocol, or it can create a new thread (or invoke an existing thread) for a *Java servlet*. The second approach avoids the overhead of creating a new process for each request. An *application server* manages several threads and provides other functionality to facilitate executing programs at the Web server's site. The additional functionality includes security, session management, and coordination of access to multiple data sources. *JavaBeans* and *Java Server Pages* are Java-based technologies that assist in creating and managing programs designed to be invoked by a Web server. **(Section 22.2)**

- *XML* is an emerging document description standard that allows us to describe the *content* and *structure* of a document in addition to giving display directives. It is based upon HTML and *SGML*, which is a powerful document description standard that is widely used. XML is designed to be simple enough to permit easy manipulation of XML documents, in contrast to SGML, while allowing users to develop their own document descriptions, unlike HTML. In particular, a DTD is a document description that is independent of the contents of a document, just like a relational database schema is a description of a database that is independent of the actual database instance. The development of DTDs for different application domains offers the potential that documents in these domains can be freely exchanged and uniformly interpreted according to standard, agreed-upon DTD descriptions. XML documents have less rigid structure than a relational database

and are said to be *semistructured*. Nonetheless, there is sufficient structure to permit many useful queries, and query languages are being developed for XML data. **(Section 22.3)**

- The proliferation of text data on the Web has brought renewed attention to information retrieval techniques for searching text. Two broad classes of search are *boolean queries* and *ranked queries*. *Boolean queries* ask for documents containing a specified boolean combination of keywords. *Ranked queries* ask for documents that are most relevant to a given list of keywords; the quality of answers is evaluated using *precision* (the percentage of retrieved documents that are relevant to the query) and *recall* (the percentage of relevant documents in the database that are retrieved) as metrics.

  Inverted files and *signature files* are two indexing techniques that support boolean queries. Inverted files are widely used and perform well, but have a high space overhead. Signature files address the space problem associated with inverted files but must be sequentially scanned. **(Section 22.4)**

- Handling ranked queries on the Web is a difficult problem. The HITS algorithm uses a combination of boolean queries and analysis of links to a page from other Web sites to evaluate ranked queries. The intuition is to find authoritative sources for the concepts listed in the query. An authoritative source is likely to be frequently cited. A good source of citations is likely to cite several good authorities. These observations can be used to assign weights to sites and identify which sites are good authorities and hubs for a given query. **(Section 22.5)**

EXERCISES

Exercise 22.1 Briefly answer the following questions.

1. Define the following terms and describe what they are used for: HTML, URL, CGI, server-side processing, Java Servlet, JavaBean, Java server page, HTML template, CCS, XML, DTD, XSL, semistructured data, inverted file, signature file.
2. What is CGI? What are the disadvantages of an architecture using CGI scripts?
3. What is the difference between a Web server and an application server? What funcionality do typical application servers provide?
4. When is an XML document well-formed? When is an XML document valid?

Exercise 22.2 Consider our bookstore example again. Assume that customers also want to search books by title.

1. Extend the HTML document shown in Figure 22.2 by another form that allows users to input the title of a book.
2. Write a Perl script similar to the Perl script shown in Figure 22.3 that generates dynamically an HTML page with all books that have the user-specified title.

Exercise 22.3 Consider the following description of items shown in the Eggface computer mail-order catalog.

"Eggface sells hardware and software. We sell the new Palm Pilot V for $400; its part number is 345. We also sell the IBM ThinkPad 570 for only $1999; choose part number 3784. We sell both business and entertainment software. Microsoft Office 2000 has just arrived and you can purchase the Standard Edition for only $140, part number 974. The new desktop publishing software from Adobe called InDesign is here for only $200, part 664. We carry the newest games from Blizzard software. You can start playing Diablo II for only $30, part number 12, and you can purchase Starcraft for only $10, part number 812."

1. Design an HTML document that depicts the items offered by Eggface.
2. Create a well-formed XML document that describes the contents of the Eggface catalog.
3. Create a DTD for your XML document and make sure that the document you created in the last question is valid with respect to this DTD.
4. Write an XML-QL query that lists all software items in the catalog.
5. Write an XML-QL query that lists the prices of all hardware items in the catalog.
6. Depict the catalog data in the semistructured data model as shown in Figure 22.8.

Exercise 22.4 A university database contains information about professors and the courses they teach. The university has decided to publish this information on the Web and you are in charge of the execution. You are given the following information about the contents of the database:

In the fall semester 1999, the course 'Introduction to Database Management Systems' was taught by Professor Ioannidis. The course took place Mondays and Wednesdays from 9–10 a.m. in room 101. The discussion section was held on Fridays from 9–10 a.m. Also in the fall semester 1999, the course 'Advanced Database Management Systems' was taught by Professor Carey. Thirty five students took that course which was held in room 110 Tuesdays and Thursdays from 1–2 p.m. In the spring semester 1999, the course 'Introduction to Database Management Systems' was taught by U.N. Owen on Tuesdays and Thursdays from 3–4 p.m. in room 110. Sixty three students were enrolled; the discussion section was on Thursdays from 4–5 p.m. The other course taught in the spring semester was 'Advanced Database Management Systems' by Professor Ioannidis, Monday, Wednesday, and Friday from 8–9 a.m.

1. Create a well-formed XML document that contains the university database.
2. Create a DTD for your XML document. Make sure that the XML document is valid with respect to this DTD.
3. Write an XML-QL query that lists the name of all professors.
4. Describe the information in a different XML document—a document that has a different structure. Create a corresponding DTD and make sure that the document is valid. Reformulate your XML-QL query that finds the names of all professors to work with the new DTD.

Exercise 22.5 Consider the database of the FamilyWear clothes manufacturer. FamilyWear produces three types of clothes: women's clothes, men's clothes, and children's clothes. Men can choose between polo shirts and T-shirts. Each polo shirt has a list of available colors, sizes, and a uniform price. Each T-shirt has a price, a list of available colors, and a list of available sizes. Women have the same choice of polo shirts and T-shirts as men. In addition women can choose between three types of jeans: slim fit, easy fit, and relaxed fit jeans. Each pair of jeans has a list of possible waist sizes and possible lengths. The price of a pair of jeans only depends on its type. Children can choose between T-shirts and baseball caps. Each T-shirt has a price, a list of available colors, and a list of available patterns. T-shirts for children all have the same size. Baseball caps come in three different sizes: small, medium, and large. Each item has an optional sales price that is offered on special occasions.

Design an XML DTD for FamilyWear so that FamilyWear can publish its catalog on the Web.

Exercise 22.6 Assume you are given a document database that contains six documents. After stemming, the documents contain the following terms:

Document	Terms
1	car manufacturer Honda auto
2	auto computer navigation
3	Honda navigation
4	manufacturer computer IBM
5	IBM personal computer
6	car Beetle VW

Answer the following questions.

1. Discuss the advantages and disadvantages of inverted files versus signature files.
2. Show the result of creating an inverted file on the documents.
3. Show the result of creating a signature file with a width of 5 bits. Construct your own hashing function that maps terms to bit positions.
4. Evaluate the following queries using the inverted file and the signature file that you created: 'car', 'IBM' `AND` 'computer', 'IBM' `AND` 'car', 'IBM' `OR` 'auto', and 'IBM' `AND` 'computer' `AND` 'manufacturer'.
5. Assume that the query load against the document database consists of exactly the queries that were stated in the previous question. Also assume that each of these queries is evaluated exactly once.
   (a) Design a signature file with a width of 3 bits and design a hashing function that minimizes the overall number of false positives retrieved when evaluating the
   (b) Design a signature file with a width of 6 bits and a hashing function that minimizes the overall number of false positives.
   (c) Assume you want to construct a signature file. What is the smallest signature width that allows you to evaluate all queries without retrieving any false positives?

Exercise 22.7 Assume that the base set of the HITS algorithm consists of the set of Web pages displayed in the following table. An entry should be interpreted as follows: Web page 1 has hyperlinks to pages 5 and 6.

Web page	Pages that this page has links to
1	5, 6, 7
2	5, 7
3	6, 8
4	
5	1, 2
6	1, 3
7	1, 2
8	4

Run five iterations of the HITS algorithm and find the highest ranked authority and the highest ranked hub.

BIBLIOGRAPHIC NOTES

The latest version of the standards mentioned in this chapter can be found from the Web pages of the World Wide Web Consortium (`www.w3.org`). Its Web site contains links to information about HTML, cascading style sheets, XML, XSL, and much more. The book by Hall is a general introduction to Web programming technologies [302]; a good starting point on the Web is `www.Webdeveloper.com`. There are many introductory books on CGI programming, for example [176, 166]. The JavaSoft (`java.sun.com`) home page is a good starting point for JavaBeans, Servlets, JSP, and all other Java-related technologies. The book by Hunter [333] is a good introduction to Java Servlets. Microsoft supports Active Server Pages (ASP), a comparable technology to JSP. More information about ASP can be found on the Microsoft Developer's Network homepage (`msdn.microsoft.com`).

There are excellent Web sites devoted to the advancement of XML, for example `www.xml.com` and `www.ibm.com/xml`, that also contain a plethora of links with information about the other standards. There are good introductory books on many different aspects of XML, for example [164, 135, 520, 411, 321, 271]. Information about UNICODE can be found on its home page `http://www.unicode.org`.

There is a lot of research on semistructured data in the database community. The Tsimmis data integration system uses a semistructured data model to cope with possible heterogeneity of data sources [509, 508]. Several new query languages for semistructured data have been developed: LOREL [525], UnQL [106], StruQL [233], and WebSQL [458]. LORE is a database management system designed for semistructured data [450]. Query optimization for semistructured data is addressed in [5, 272]. Work on describing the structure of semistructured databases can be found in [490, 272].

There has been a lot of work on using semistructured data models for Web data and several Web query systems have been developed: WebSQL [458], W3QS [384], WebLog [399], WebOQL [32], STRUDEL [232], ARANEUS [39], and FLORID [319]. [237] is a good overview of database research in the context of the Web.

Introductory reading material on information retrieval includes the standard textbooks by Salton and McGill [562] and by van Rijsbergen [661]. Collections of articles for the more

advanced reader have been edited by Jones and Willett [350] and by Frakes and Baeza-Yates [239]. Querying text repositories has been studied extensively in information retrieval; see [545] for a recent survey. Faloutsos overviews indexing methods for text databases [219]. Inverted files are discussed in [469] and signature files are discussed in [221]. Zobel, Moffat, and Ramamohanarao give a comparison of inverted files and signature files [703]. Other aspects of indexing for text databases are addressed in [704]. The book by Witten, Moffat, and Bell has a lot of material on compression techniques for document databases [685].

The number of citation counts as a measure of scientific impact has first been studied by Garfield [262]; see also [670]. Usage of hypertextual information to improve the quality of search engines has been proposed by Spertus [610] and by Weiss et al. [676]. The HITS algorithm was developed by Jon Kleinberg [376]. Concurrently, Brin and Page developed the Pagerank algorithm, which also takes hyperlinks between pages into account [99]. The discovery of structure in the World Wide Web is currently a very active area of research; see for example the work by Gibson et al. [268].

24

DATA MINING

The secret of success is to know something nobody else knows.

—Aristotle Onassis

Data mining consists of finding interesting trends or patterns in large datasets, in order to guide decisions about future activities. There is a general expectation that data mining tools should be able to identify these patterns in the data with minimal user input. The patterns identified by such tools can give a data analyst useful and unexpected insights that can be more carefully investigated subsequently, perhaps using other decision support tools. In this chapter, we discuss several widely studied data mining tasks. There are commercial tools available for each of these tasks from major vendors, and the area is rapidly growing in importance as these tools gain acceptance from the user community.

We start in Section 24.1 by giving a short introduction to data mining. In Section 24.2, we discuss the important task of counting co-occurring items. In Section 24.3, we discuss how this task arises in data mining algorithms that discover rules from the data. In Section 24.4, we discuss patterns that represent rules in the form of a tree. In Section 24.5, we introduce a different data mining pattern called clustering and consider how to find clusters. In Section 24.6, we consider how to perform similarity search over sequences. We conclude with a short overview of other data mining tasks in Section 24.7.

24.1 INTRODUCTION TO DATA MINING

Data mining is related to the subarea of statistics called *exploratory data analysis*, which has similar goals and relies on statistical measures. It is also closely related to the subareas of artificial intelligence called *knowledge discovery* and *machine learning*. The important distinguishing characteristic of data mining is that the volume of data is very large; although ideas from these related areas of study are applicable to data mining problems, *scalability with respect to data size* is an important new criterion. An algorithm is **scalable** if the running time grows (linearly) in proportion to the dataset size, given the available system resources (e.g., amount of main memory and disk). Old algorithms must be adapted or new algorithms must be developed to ensure scalability.

Finding useful trends in datasets is a rather loose definition of data mining: In a certain sense, all database queries can be thought of as doing just this. Indeed, we have a continuum of analysis and exploration tools with SQL queries at one end, OLAP queries in the middle, and data mining techniques at the other end. SQL queries are constructed using relational algebra (with some extensions); OLAP provides higher-level querying idioms based on the multidimensional data model; and data mining provides the most abstract analysis operations. We can think of different data mining tasks as complex 'queries' specified at a high level, with a few parameters that are user-definable, and for which specialized algorithms are implemented.

In the real world, data mining is much more than simply applying one of these algorithms. Data is often noisy or incomplete, and unless this is understood and corrected for, it is likely that many interesting patterns will be missed and the reliability of detected patterns will be low. Further, the analyst must decide what kinds of mining algorithms are called for, apply them to a well-chosen subset of data samples and variables (i.e., tuples and attributes), digest the results, apply other decision support and mining tools, and iterate the process.

The **knowledge discovery process**, or short **KDD process**, can roughly be separated into five steps. The raw data first undergoes a **data selection** step, in which we identify the target dataset and relevant attributes. Then in a **data cleaning** step, we remove noise and outliers, transform field values to common units, generate new fields through combination of existing fields, and bring the data into the relational schema that is used as input to the data mining activity. The data cleaning step might also involve a denormalization of the underlying relations. In the **data mining** step, we extract the actual patterns. In the final step, the **evaluation** step, we present the patterns in an understandable form to the end user, for example through visualization. The results of any step in the KDD process might lead us back to an earlier step in order to redo the process with the new knowledge gained. In this chapter, however, we will limit ourselves to looking at algorithms for some specific data mining tasks. We will not discuss other aspects of the KDD process further.

24.2 COUNTING CO-OCCURRENCES

We begin by considering the problem of counting co-occurring items, which is motivated by problems such as market basket analysis. A **market basket** is a collection of items purchased by a customer in a single **customer transaction**. A customer transaction consists of a single visit to a store, a single order through a mail-order catalog, or an order at a virtual store on the web. (In this chapter, we will often abbreviate *customer transaction* by *transaction* when there is no confusion with the usual meaning of *transaction* in a DBMS context, which is an execution of a user program.) A common goal for retailers is to identify items that are purchased together. This information can be used to improve the layout of goods in a store or the layout of catalog pages.

transid	*custid*	*date*	*item*	*qty*
111	201	5/1/99	pen	2
111	201	5/1/99	ink	1
111	201	5/1/91	milk	3
111	201	5/1/99	juice	6
112	105	6/3/99	pen	1
112	105	6/3/99	ink	1
112	105	6/3/99	milk	1
113	106	5/10/99	pen	1
113	106	5/10/99	milk	1
114	201	6/1/99	pen	2
114	201	6/1/99	ink	2
114	201	6/1/99	juice	4

Figure 24.1 The Purchases Relation for Market Basket Analysis

24.2.1 Frequent Itemsets

We will use the Purchases relation shown in Figure 24.1 to illustrate frequent itemsets. The records are shown sorted into groups by transaction. All tuples in a group have the same *transid*, and together they describe a customer transaction, which involves purchases of one or more items. A transaction occurs on a given date, and the name of each purchased item is recorded, along with the purchased quantity. Observe that there is redundancy in Purchases: It can be decomposed by storing *transid–custid* pairs separately and dropping *custid*; this may be how the data is actually stored. However, it is convenient to consider the Purchases relation as it is shown in Figure 24.1 in order to compute frequent itemsets. Creating such 'denormalized' tables for ease of data mining is commonly done in the data cleaning step of the KDD process.

By examining the set of transaction groups in Purchases, we can make observations of the form: "In 75 percent of the transactions both a pen and ink are purchased together." It is a statement that describes the transactions in the database. Extrapolation to future transactions should be done with caution, as discussed in Section 24.3.6. Let us begin by introducing the terminology of market basket analysis. An **itemset** is a set of items. The **support** of an itemset is the fraction of transactions in the database that contain all the items in the itemset. In our example, we considered the itemset {pen, ink} and observed that the support of this itemset was 75 percent in Purchases. We can thus conclude that pens and ink are frequently purchased together. If we consider the itemset {milk, juice}, its support is only 25 percent. Thus milk and juice are not purchased together frequently.

Usually the number of sets of items that are frequently purchased together is relatively small, especially as the size of the itemsets increases. We are interested in all itemsets whose support is higher than a user-specified minimum support called *minsup*; we call such itemsets **frequent itemsets**. For example, if the minimum support is set to 70 percent, then the frequent itemsets in our example are {pen}, {ink}, {milk}, {pen, ink}, and {pen, milk}. Note that we are also interested in itemsets that contain only a single item since they identify items that are purchased frequently.

We show an algorithm for identifying frequent itemsets in Figure 24.2. This algorithm relies upon a simple yet fundamental property of frequent itemsets:

> **The a priori property**: Every subset of a frequent itemset must also be a frequent itemset.

The algorithm proceeds iteratively, first identifying frequent itemsets with just one item. In each subsequent iteration, frequent itemsets identified in the previous iteration are extended with another item to generate larger candidate itemsets. By considering only itemsets obtained by enlarging frequent itemsets, we greatly reduce the number of candidate frequent itemsets; this optimization is crucial for efficient execution. The a priori property guarantees that this optimization is correct, that is, we don't miss any frequent itemsets. A single scan of all transactions (the Purchases relation in our example) suffices to determine which candidate itemsets generated in an iteration are frequent itemsets. The algorithm terminates when no new frequent itemsets are identified in an iteration.

```
foreach item,                                              // Level 1
    check if it is a frequent itemset          // appears in > minsup transactions
k = 1
repeat                    // Iterative, level-wise identification of frequent itemsets
    foreach new frequent itemset I_k with k items                  // Level k + 1
        generate all itemsets I_{k+1} with k + 1 items, I_k ⊂ I_{k+1}
    Scan all transactions once and check if
    the generated k + 1-itemsets are frequent
    k = k + 1
until no new frequent itemsets are identified
```

Figure 24.2 An Algorithm for Finding Frequent Itemsets

We illustrate the algorithm on the Purchases relation in Figure 24.1, with *minsup* set to 70 percent. In the first iteration (Level 1), we scan the Purchases relation and determine that each of these one-item sets is a frequent itemset: {*pen*} (appears in all four transactions), {*ink*} (appears in three out of four transactions), and {*milk*} (appears in three out of four transactions).

In the second iteration (Level 2), we extend each frequent itemset with an additional item and generate the following candidate itemsets: {*pen, ink*}, {*pen, milk*}, {*pen, juice*}, {*ink, milk*}, {*ink, juice*}, and {*milk, juice*}. By scanning the Purchases relation again, we determine that the following are frequent itemsets: {*pen, ink*} (appears in three out of four transactions), and {*pen, milk*} (appears in three out of four transactions).

In the third iteration (Level 3), we extend these itemsets with an additional item, and generate the following candidate itemsets: {*pen, ink, milk*}, {*pen, ink, juice*}, and {*pen, milk, juice*}. (Observe that {*ink, milk, juice*} is not generated.) A third scan of the Purchases relation allows us to determine that none of these is a frequent itemset.

The simple algorithm presented here for finding frequent itemsets illustrates the principal feature of more sophisticated algorithms, namely the iterative generation and testing of candidate itemsets. We consider one important refinement of this simple algorithm. Generating candidate itemsets by adding an item to an itemset that is already known to be frequent is an attempt to limit the number of candidate itemsets using the a priori property. The a priori property implies that a candidate itemset can only be frequent if all its subsets are frequent. Thus, we can reduce the number of candidate itemsets further—a priori to scanning the Purchases database—by checking whether all subsets of a newly generated candidate itemset are frequent. Only if all subsets of a candidate itemset are frequent do we compute its support in the subsequent database scan. Compared to the simple algorithm, this refined algorithm generates fewer candidate itemsets at each level and thus reduces the amount of computation performed during the database scan of Purchases.

Consider the refined algorithm on the Purchases table in Figure 24.1 with *minsup*= 70 percent. In the first iteration (Level 1), we determine the frequent itemsets of size one: {*pen*}, {*ink*}, and {*milk*}. In the second iteration (Level 2), only the following candidate itemsets remain when scanning the Purchases table: {*pen, ink*}, {*pen, milk*}, and {*ink, milk*}. Since {*juice*} is not frequent, the itemsets {*pen, juice*}, {*ink, juice*}, and {*milk, juice*} cannot be frequent as well and we can eliminate those itemsets a priori, that is, without considering them during the subsequent scan of the Purchases relation. In the third iteration (Level 3), no further candidate itemsets are generated. The itemset {*pen, ink, milk*} cannot be frequent since its subset {*ink, milk*} is not frequent. Thus, the improved version of the algorithm does not need a third scan of Purchases.

24.2.2 Iceberg Queries

We introduce iceberg queries through an example. Consider again the Purchases relation shown in Figure 24.1. Assume that we want to find pairs of customers and items

such that the customer has purchased the item at least twice. We can express this query in SQL as follows:

```
SELECT   P.custid, P.item, SUM (P.qty)
FROM     Purchases P
GROUP BY P.custid, P.item
HAVING   SUM (P.qty) > 5
```

Think about how this query would be evaluated by a relational DBMS. Conceptually, for each (*custid, item*) pair, we need to check whether the sum of the *qty* field is greater than 5. One approach is to make a scan over the Purchases relation and maintain running sums for each (*custid,item*) pair. This is a feasible execution strategy as long as the number of pairs is small enough to fit into main memory. If the number of pairs is larger than main memory, more expensive query evaluation plans that involve either sorting or hashing have to be used.

The query has an important property that is not exploited by the above execution strategy: Even though the Purchases relation is potentially very large and the number of (*custid,item*) groups can be huge, the output of the query is likely to be relatively small because of the condition in the HAVING clause. Only groups where the customer has purchased the item more than five times appear in the output. For example, there are nine groups in the query over the Purchases relation shown in Figure 24.1, although the output only contains three records. The number of groups is very large, but the answer to the query—the tip of the iceberg—is usually very small. Therefore, we call such a query an **iceberg query**. In general, given a relational schema R with attributes A1, A2, ..., Ak, and B and an aggregation function aggr, an iceberg query has the following structure:

```
SELECT   R.A1, R.A2, ..., R.Ak, aggr(R.B)
FROM     Relation R
GROUP BY R.A1, ..., R.Ak
HAVING   aggr(R.B) >= constant
```

Traditional query plans for this query that use sorting or hashing first compute the value of the aggregation function for all groups and then eliminate groups that do not satisfy the condition in the HAVING clause.

Comparing the query with the problem of finding frequent itemsets that we discussed in the previous section, there is a striking similarity. Consider again the Purchases relation shown in Figure 24.1 and the iceberg query from the beginning of this section. We are interested in (*custid, item*) pairs that have SUM (P.qty) > 5. Using a variation of the a priori property, we can argue that we only have to consider values of the *custid* field where the customer has purchased at least five items overall. We can generate such items through the following query:

```
SELECT    P.custid
FROM      Purchases P
GROUP BY  P.custid
HAVING    SUM (P.qty) > 5
```

Similarly, we can restrict the candidate values for the *item* field through the following query:

```
SELECT    P.item
FROM      Purchases P
GROUP BY  P.item
HAVING    SUM (P.qty) > 5
```

If we restrict the computation of the original iceberg query to (*custid*, *item*) groups where the field values are in the output of the previous two queries, we eliminate a large number of (*custid*, *item*) pairs a priori! Thus, a possible evaluation strategy is to first compute candidate values for the *custid* and *item* fields, and only to use combinations of these values in the evaluation of the original iceberg query. We first generate candidate field values for individual fields and only use those values that survive the a priori pruning step as expressed in the two previous queries. Thus, the iceberg query is amenable to the same bottom-up evaluation strategy that is used to find frequent itemsets. In particular, we can use the a priori property as follows: We only keep a counter for a group if each individual component of the group satisfies the condition expressed in the `HAVING` clause. The performance improvements of this alternative evaluation strategy over traditional query plans can be very significant in practice.

Even though the bottom-up query processing strategy eliminates many groups a priori, the number of (*custid*, *item*) pairs can still be very large in practice—even larger than main memory. Efficient strategies that use sampling and more sophisticated hashing techniques have been developed; the references at the end of the chapter provide pointers to the relevant literature.

24.3 MINING FOR RULES

Many algorithms have been proposed for discovering various forms of rules that succinctly describe the data. We now look at some widely discussed forms of rules and algorithms for discovering them.

24.3.1 Association Rules

We will use the Purchases relation shown in Figure 24.1 to illustrate association rules. By examining the set of transactions in Purchases, we can identify rules of the form:

$$\{pen\} \Rightarrow \{ink\}$$

This rule should be read as follows: "If a pen is purchased in a transaction, it is likely that ink will also be purchased in that transaction." It is a statement that describes the transactions in the database; extrapolation to future transactions should be done with caution, as discussed in Section 24.3.6. More generally, an **association rule** has the form $LHS \Rightarrow RHS$, where both LHS and RHS are sets of items. The interpretation of such a rule is that if every item in LHS is purchased in a transaction, then it is likely that the items in RHS are purchased as well.

There are two important measures for an association rule:

- **Support:** The support for a set of items is the percentage of transactions that contain all of these items. The support for a rule $LHS \Rightarrow RHS$ is the support for the set of items $LHS \cup RHS$. For example, consider the rule $\{pen\} \Rightarrow \{ink\}$. The support of this rule is the support of the itemset $\{pen, ink\}$, which is 75 percent.
- **Confidence:** Consider transactions that contain all items in LHS. The confidence for a rule $LHS \Rightarrow RHS$ is the percentage of such transactions that also contain all items in RHS. More precisely, let $sup(LHS)$ be the percentage of transactions that contain LHS and let $sup(LHS \cup RHS)$ be the percentage of transactions that contain both LHS and RHS. Then the confidence of the rule $LHS \Rightarrow RHS$ is $sup(LHS)$ / $sup(LHS \cup RHS)$. The confidence of a rule is an indication of the strength if the rule. As an example, consider again the rule $\{pen\} \Rightarrow \{ink\}$. The confidence of this rule is 75 percent; 75 percent of the transactions that contain the itemset $\{pen\}$ also contain the itemset $\{ink\}$.

24.3.2 An Algorithm for Finding Association Rules

A user can ask for all association rules that have a specified minimum support *(minsup)* and minimum confidence *(minconf)*, and various algorithms have been developed for finding such rules efficiently. These algorithms proceed in two steps. In the first step, all frequent itemsets with the user-specified minimum support are computed. In the second step, rules are generated using the frequent itemsets as input. We discussed an algorithm for finding frequent itemsets in Section 24.2, thus we concentrate here on the rule generation part.

Once frequent itemsets are identified, the generation of all possible candidate rules with the user-specified minimum support is straightforward. Consider a frequent itemset

X with support s_X identified in the first step of the algorithm. To generate a rule from X, we divide X into two itemsets *LHS* and *RHS*. The confidence of the rule *LHS* $\Rightarrow$ *RHS* is s_X/s_{LHS}, the ratio of the support of X and the support of *LHS*. From the a priori property, we know that the support of *LHS* is larger than *minsup*, and thus we have computed the support of *LHS* during the first step of the algorithm. We can compute the confidence values for the candidate rule by calculating the ratio support(X)/ support(*LHS*) and then check how the ratio compares to *minconf*.

In general, the expensive step of the algorithm is the computation of the frequent itemsets and many different algorithms have been developed to perform this step efficiently. Rule generation—given that all frequent itemsets have been identified—is straightforward.

In the remainder of this section we will discuss some generalizations of the problem.

24.3.3 Association Rules and ISA Hierarchies

In many cases an **ISA hierarchy** or **category hierarchy** is imposed upon the set of items. In the presence of a hierarchy, a transaction contains for each of its items implicitly all the item's ancestors in the hierarchy. For example, consider the category hierarchy shown in Figure 24.3. Given this hierarchy, the Purchases relation is conceptually enlarged by the eight records shown in Figure 24.4. That is, the Purchases relation has all tuples shown in Figure 24.1 in addition to the tuples shown in Figure 24.4.

The hierarchy allows us to detect relationships between items at different levels of the hierarchy. As an example, the support of the itemset {*ink*, *juice*} is 50 percent, but if we replace *juice* with the more general category *beverage*, the support of the resulting itemset {*ink*, *beverage*} increases to 75 percent. In general, the support of an itemset can only increase if an item is replaced by one of its ancestors in the ISA hierarchy.

Assuming that we actually physically add the eight records shown in Figure 24.4 to the Purchases relation, we can use any algorithm for computing frequent itemsets on the augmented database. Assuming that the hierarchy fits into main memory, we can also perform the addition on-the-fly while we are scanning the database, as an optimization.



Figure 24.3 An ISA Category Taxonomy

transid	*custid*	*date*	*item*	*qty*
111	201	5/1/99	stationery	3
111	201	5/1/99	beverage	9
112	105	6/3/99	stationery	2
112	105	6/3/99	beverage	1
113	106	5/1/99	stationery	1
113	106	5/1/99	beverage	1
114	201	5/15/99	stationery	4
114	201	5/15/99	beverage	4

Figure 24.4 Conceptual Additions to the Purchases Relation with ISA Hierarchy

24.3.4 Generalized Association Rules

Although association rules have been most widely studied in the context of market basket analysis, or analysis of customer transactions, the concept is more general. Consider the Purchases relation as shown in Figure 24.5, grouped by *custid*. By examining the set of customer groups, we can identify association rules such as $\{\text{pen}\} \Rightarrow \{\text{ink}\}$. This rule should now be read as follows: "If a pen is purchased by a customer, it is likely that milk will also be purchased by that customer." In the Purchases relation shown in Figure 24.5, this rule has both support and confidence of 100 percent.

transid	*custid*	*date*	*item*	*qty*
112	105	6/3/99	pen	1
112	105	6/3/99	ink	1
112	105	6/3/99	milk	1
113	106	5/1/99	pen	1
113	106	5/1/99	milk	1
114	201	5/15/99	pen	2
114	201	5/15/99	ink	2
114	201	5/15/99	juice	4
111	201	5/1/99	pen	2
111	201	5/1/99	ink	1
111	201	5/1/99	milk	3
111	201	5/1/99	juice	6

Figure 24.5 The Purchases Relation Sorted on Customer Id

Similarly, we can group tuples by date and identify association rules that describe purchase behavior on the same day. As an example consider again the Purchases relation. In this case, the rule {pen} ⇒ {ink} is now interpreted as follows: "On a day where a pen is purchased, it is likely that milk will also be purchased as well."

If we use the *date* field as grouping attribute, we can consider a more general problem called **calendric market basket analysis**. In calendric market basket analysis, the user specifies a collection of **calendars**. A calendar is any group of dates, e.g., *every Sunday in the year 1999,* or *every first of the month.* A rule holds if it holds on every day in the calendar. Given a calendar, we can compute association rules over the set of tuples whose *date* field falls within the calendar.

By specifying interesting calendars, we can identify rules that might not have enough support and confidence with respect to the entire database, but that have enough support and confidence on the subset of tuples that fall within the calendar. On the other hand, even though a rule might have enough support and confidence with respect to the complete database, it might gain its support only from tuples that fall within a calendar. In this case, the support of the rule over the tuples within the calendar is significantly higher than its support with respect to the entire database.

As an example, consider the Purchases relation with the calendar *every first of the month.* Within this calendar, the association rule *pen* ⇒ *juice* has support and confidence of 100 percent, whereas over the entire Purchases relation, this rule only has 50 percent support. On the other hand, within the calendar, the rule *pen* ⇒ *milk* has support of confidence of 50 percent, whereas over the entire Purchases relation it has support and confidence of 75 percent.

More general specifications of the conditions that must be true within a group for a rule to hold (for that group) have also been proposed. We might want to say that all items in the *LHS* have to be purchased in a quantity of less than two items, and all items in the *RHS* must be purchased in a quantity of more than three.

Using different choices for the grouping attribute and sophisticated conditions as in the above examples, we can identify rules that are more complex than the basic association rules discussed earlier. These more complex rules, nonetheless, retain the essential structure of an association rule as a condition over a group of tuples, with support and confidence measures defined as usual.

24.3.5 Sequential Patterns

Consider the Purchases relation shown in Figure 24.1. Each group of tuples, having the same *custid* value, can be thought of as a *sequence* of transactions ordered by *date.* This allows us to identify frequently arising buying patterns over time.

We begin by introducing the concept of a sequence of itemsets. Each transaction is represented by a set of tuples, and by looking at the values in the *item* column, we get a set of items purchased in that transaction. Thus, the sequence of transactions associated with a customer corresponds naturally to a sequence of itemsets purchased by the customer. For example, the sequence of purchases for customer 201 is {pen,ink,milk,juice}, {pen,ink,juice}.

A **subsequence** of a sequence of itemsets is obtained by deleting one or more itemsets, and is also a sequence of itemsets. We say that a sequence $\{a_1, \ldots a_m\}$ is **contained** in another sequence S if S has a subsequence $\{b_1, \ldots b_m\}$ such that $a_i \subseteq b_i$, for $1 \leq i \leq m$. Thus, the sequence {pen}, {ink,milk}, {pen,juice} is contained in {pen,ink}, {shirt}, {juice,ink,milk}, {juice,pen,milk}. Note that the order of items within each itemset does not matter. However, the order of itemsets does matter: the sequence {pen}, {ink,milk}, {pen,juice} is not contained in {pen,ink}, {shirt}, {juice,pen,milk}, {juice,milk,ink}.

The **support** for a sequence S of itemsets is the percentage of customer sequences of which S is a subsequence. The problem of identifying sequential patterns is to find all sequences that have a user-specified minimum support. A sequence $s_1, s_2, s_3, \ldots, s_n$ with minimum support tells us that customers often purchase the items in set s_1 in a transaction, then in some subsequent transaction buy the items in set s_2, then the items in set s_3 in a later transaction, and so on.

Like association rules, sequential patterns are statements about groups of tuples in the current database. Computationally, algorithms for finding frequently occurring sequential patterns resemble algorithms for finding frequent itemsets. Longer and longer sequences with the required minimum support are identified iteratively in a manner that is very similar to the iterative identification of frequent itemsets.

24.3.6 The Use of Association Rules for Prediction

Association rules are widely used for prediction, but it is important to recognize that such predictive use is not justified without additional analysis or domain knowledge. Association rules describe existing data accurately but can be misleading when used naively for prediction. For example, consider the rule

$$\{pen\} \Rightarrow \{ink\}$$

The confidence associated with this rule is the conditional probability of an ink purchase given a pen purchase *over the given database*; that is, it is a *descriptive* measure. We might use this rule to guide future sales promotions. For example, we might offer a discount on pens in order to increase the sales of pens and, therefore, also increase sales of ink.

However, such a promotion assumes that pen purchases are good indicators of ink purchases in *future* customer transactions (in addition to transactions in the current database). This assumption is justified if there is a *causal link* between pen purchases and ink purchases; that is, if buying pens causes the buyer to also buy ink. However, we can infer association rules with high support and confidence in some situations where there is no causal link between *LHS* and *RHS*! For example, suppose that pens are always purchased together with pencils, perhaps because of customers' tendency to order writing instruments together. We would then infer the rule

$$\{pencil\} \Rightarrow \{ink\}$$

with the same support and confidence as the rule

$$\{pen\} \Rightarrow \{ink\}$$

However, there is no causal link between pencils and ink. If we promote pencils, a customer who purchases several pencils due to the promotion has no reason to buy more ink. Thus, a sales promotion that discounted pencils in order to increase the sales of ink would fail.

In practice, one would expect that by examining a large database of past transactions (collected over a long time and a variety of circumstances) and restricting attention to rules that occur often (i.e., that have high support), we minimize inferring misleading rules. However, we should bear in mind that misleading, noncausal rules will be generated. Therefore, we should treat the generated rules as possibly, rather than conclusively, identifying causal relationships. Although association rules do not indicate causal relationships between the *LHS* and *RHS*, we emphasize that they provide a useful starting point for identifying such relationships, using either further analysis or a domain expert's judgment; this is the reason for their popularity.

24.3.7 Bayesian Networks

Finding causal relationships is a challenging task, as we saw in Section 24.3.6. In general, if certain events are highly correlated, there are many possible explanations. For example, suppose that pens, pencils, and ink are purchased together frequently. It might be the case that the purchase of one of these items (e.g., ink) depends causally upon the purchase of another item (e.g., pen). Or it might be the case that the purchase of one of these items (e.g., pen) is strongly correlated with the purchase of another (e.g., pencil) because there is some underlying phenomenon (e.g., users' tendency to think about writing instruments together) that causally influences both purchases. How can we identify the true causal relationships that hold between these events in the real world?

One approach is to consider each possible combination of causal relationships among the variables or events of interest to us and to evaluate the likelihood of each combina-

tion on the basis of the data available to us. If we think of each combination of causal relationships as a *model* of the real world underlying the collected data, we can assign a score to each model by considering how consistent it is (in terms of probabilities, with some simplifying assumptions) with the observed data. Bayesian networks are graphs that can be used to describe a class of such models, with one node per variable or event, and arcs between nodes to indicate causality. For example, a good model for our running example of pens, pencils, and ink is shown in Figure 24.6. In general, the number of possible models is exponential in the number of variables, and considering all models is expensive, so some subset of all possible models is evaluated.

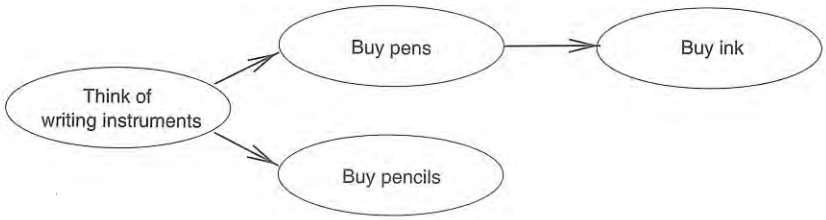


Figure 24.6 Bayesian Network Showing Causality

24.3.8 Classification and Regression Rules

Consider the following view that contains information from a mailing campaign performed by a publishing company:

InsuranceInfo(*age:* `integer`, *cartype:* `string`, *highrisk:* `boolean`)

The InsuranceInfo view has information about current customers. Each record contains a customer's age and type of car as well as a flag indicating whether the person is considered a high-risk customer. If the flag is true, the customer is considered high-risk. We would like to use this information to identify rules that predict the insurance risk of new insurance applicants whose age and car type are known. For example, one such rule could be: "If *age* is between 16 and 25 and *cartype* is either Sports or Truck, then the risk is high."

Note that the rules we want to find have a specific structure. We are not interested in rules that predict the age or type of car of a person; we are only interested in rules that predict the insurance risk. Thus, there is one designated attribute whose value we would like to predict and we will call this attribute the **dependent** attribute. The other attributes are called **predictor** attributes. In our example, the dependent attribute in the InsuranceInfo view is the *highrisk* attribute and the predictor attributes are *age* and *cartype*. The general form of the types of rules we want to discover is:

$$P_1(X_1) \wedge P_2(X_2) \ldots \wedge P_k(X_k) \Rightarrow Y = c$$

The predictor attributes $X_1, \ldots, X_k$ are used to predict the value of the dependent attribute Y. Both sides of a rule can be interpreted as conditions on fields of a tuple. The $P_i(X_i)$ are predicates that involve attribute X_i. The form of the predicate depends on the type of the predictor attribute. We distinguish two types of attributes: **numerical** and **categorical** attributes. For numerical attributes, we can perform numerical computations such as computing the average of two values, whereas for categorical attributes their domain is a set of values. In the InsuranceInfo view, *age* is a numerical attribute whereas *cartype* and *highrisk* are categorical attributes. Returning to the form of the predicates, if X_i is a numerical attribute, its predicate P_i is of the form $li \leq X_i \leq hi$; if X_i is a categorical attribute, P_i is of the form $X_i \in \{v_1, \ldots, v_j\}$.

If the dependent attribute is categorical, we call such rules **classification rules**. If the dependent attribute is numerical, we call such rules **regression rules**.

For example, consider again our example rule: "If *age* is between 16 and 25 and *cartype* is either Sports or Truck, then *highrisk* is true." Since *highrisk* is a categorical attribute, this rule is a classification rule. We can express this rule formally as follows:

$$(16 \leq age \leq 25) \wedge (cartype \in \{\text{Sports, Truck}\}) \Rightarrow highrisk = \texttt{true}$$

We can define support and confidence for classification and regression rules, as for association rules:

- **Support:** The support for a condition C is the percentage of tuples that satisfy C. The support for a rule $C1 \Rightarrow C2$ is the support for the condition $C1 \wedge C2$.
- **Confidence:** Consider those tuples that satisfy condition $C1$. The confidence for a rule $C1 \Rightarrow C2$ is the percentage of such tuples that also satisfy condition $C2$.

As a further generalization, consider the right-hand side of a classification or regression rule: $Y = c$. Each rule predicts a value of Y for a given tuple based on the values of predictor attributes $X1, \ldots, Xk$. We can consider rules of the form:

$$P_1(X_1) \wedge \ldots \wedge P_k(X_k) \Rightarrow Y = f(X_1, \ldots, X_k)$$

where f is some function. We will not discuss such rules further.

Classification and regression rules differ from association rules by considering continuous and categorical fields, rather than only one field that is set-valued. Identifying such rules efficiently presents a new set of challenges and we will not discuss the general case of discovering such rules. We will discuss a special type of such rules in Section 24.4.

Classification and regression rules have many applications. Examples include classification of results of scientific experiments, where the type of object to be recognized

depends on the measurements taken; direct mail prospecting, where the response of a given customer to a promotion is a function of his or her income level and age; and car insurance risk assessment, where a customer could be classified as risky depending on his age, profession, and car type. Example applications of regression rules include financial forecasting, where the price of coffee futures could be some function of the rainfall in Colombia a month ago, and medical prognosis, where the likelihood of a tumor being cancerous is a function of measured attributes of the tumor.

24.4 TREE-STRUCTURED RULES

In this section, we discuss the problem of discovering classification and regression rules from a relation, but we consider only rules that have a very special structure. The type of rules that we discuss can be represented by a tree, and typically the tree itself is the output of the data mining activity. Trees that represent classification rules are called **classification trees** or **decision trees** and trees that represent regression rules are called **regression trees**.

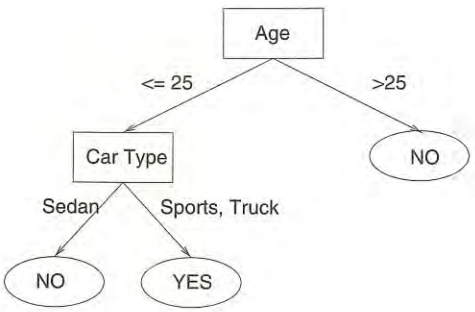


Figure 24.7 Insurance Risk Example Decision Tree

As an example, consider the decision tree shown in Figure 24.7. Each path from the root node to a leaf node represents one classification rule. For example, the path from the root to the leftmost leaf node represents the classification rule: "If a person is 25 years or younger and drives a sedan, then he is likely to have a low insurance risk." The path from the root to the right-most leaf node represents the classification rule: "If a person is older than 25 years, then he is likely to have a low insurance risk."

Tree-structured rules are very popular since they are easy to interpret. Ease of understanding is very important, since the result of any data mining activity needs to be comprehensible by nonspecialists. In addition, studies have shown that despite limitations in structure, tree-structured rules are very accurate. There exist efficient

algorithms to construct tree-structured rules from large databases. We will discuss a sample algorithm for decision tree construction in the remainder of this section.

24.4.1 Decision Trees

A decision tree is a graphical representation of a collection of classification rules. Given a data record, the tree directs the record from the root to a leaf. Each internal node of the tree is labeled with a predictor attribute. This attribute is often called a **splitting attribute**, because the data is 'split' based on conditions over this attribute. The outgoing edges of an internal node are labeled with predicates that involve the splitting attribute of the node; every data record entering the node must satisfy the predicate labeling exactly one outgoing edge. The combined information about the splitting attribute and the predicates on the outgoing edges is called the **splitting criterion** of the node. A node with no outgoing edges is called a **leaf node**. Each leaf node of the tree is labeled with a value of the dependent attribute. We only consider binary trees where internal nodes have two outgoing edges, although trees of higher degree are possible.

Consider the decision tree shown in Figure 24.7. The splitting attribute of the root node is *age*, the splitting attribute of the left child of the root node is *cartype*. The predicate on the left outgoing edge of the root node is $age \leq 25$, the predicate on the right outgoing edge is $age > 25$.

We can now associate a classification rule with each leaf node in the tree as follows. Consider the path from the root of the tree to the leaf node. Each edge on that path is labeled with a predicate. The conjunction of all these predicates makes up the left hand side of the rule. The value of the dependent attribute at the leaf node makes up the right-hand side of the rule. Thus, the decision tree represents a collection of classification rules, one for each leaf node.

A decision tree is usually constructed in two phases. In phase one, the **growth phase**, an overly large tree is constructed. This tree represents the records in the input database very accurately; for example, the tree might contain leaf nodes for individual records from the input database. In phase two, the **pruning phase**, the final size of the tree is determined. The rules represented by the tree constructed in phase one are usually overspecialized. By reducing the size of the tree, we generate a smaller number of more general rules that are better than a very large number of very specialized rules. Algorithms for tree pruning are beyond our scope of discussion here.

Classification tree algorithms build the tree greedily top-down in the following way. At the root node, the database is examined and the locally 'best' splitting criterion is computed. The database is then partitioned, according to the root node's splitting criterion, into two parts, one partition for the left child and one partition for the

Input: node n, partition D, split selection method $\mathcal{S}$
Output: decision tree for D rooted at node n

Top-Down Decision Tree Induction Schema:
BuildTree(Node n, data partition D, split selection method $\mathcal{S}$)
(1) Apply $\mathcal{S}$ to D to find the splitting criterion
(2) **if** (a good splitting criterion is found)
(3) Create two children nodes n_1 and n_2 of n
(4) Partition D into D_1 and D_2
(5) BuildTree(n_1, D_1, $\mathcal{S}$)
(6) BuildTree(n_2, D_2, $\mathcal{S}$)
(7) **endif**

Figure 24.8 Decision Tree Induction Schema

right child. The algorithm then recurses on each child. This schema is depicted in Figure 24.8.

The splitting criterion at a node is found through application of a **split selection method**. A split selection method is an algorithm that takes as input (part of) a relation and outputs the locally 'best' splitting criterion. In our example, the split selection method examines the attributes *cartype* and *age*, selects one of them as splitting attribute, and then selects the splitting predicates. Many different, very sophisticated split selection methods have been developed; the references provide pointers to the relevant literature.

age	*cartype*	*highrisk*
23	Sedan	false
30	Sports	false
36	Sedan	false
25	Truck	true
30	Sedan	false
23	Truck	true
30	Truck	false
25	Sports	true
18	Sedan	false

Figure 24.9 The InsuranceInfo Relation

24.4.2 An Algorithm to Build Decision Trees

If the input database fits into main memory we can directly follow the classification tree induction schema shown in Figure 24.8. How can we construct decision trees when the input relation is larger than main memory? In this case, step (1) in Figure 24.8 fails, since the input database does not fit in memory. But we can make one important observation about split selection methods that helps us to reduce the main memory requirements.

Consider a node of the decision tree. The split selection method has to make two decisions after examining the partition at that node: (i) It has to select the splitting attribute, and (ii) It has to select the splitting predicates for the outgoing edges. Once decided on the splitting criterion, the algorithm is recursively applied to each of the children of the node. Does a split selection method actually need the complete database partition as input? Fortunately, the answer is no.

Split selection methods that compute splitting criteria that involve a single predictor attribute at each node evaluate each predictor attribute individually. Since each attribute is examined separately, we can provide the split selection method with aggregated information about the database instead of loading the complete database into main memory. Chosen correctly, this aggregated information is sufficient to compute the 'best' splitting criterion—the same splitting criterion as if the complete database would reside in main memory.

Since the split selection method examines all predictor attributes, we need aggregated information about each predictor attribute. We call this aggregated information the **AVC set** of the predictor attribute. The AVC set of a predictor attribute X at node n is the projection of n's database partition onto X and the dependent attribute where counts of the individual values in the domain of the dependent attribute are aggregated. (The acronym AVC stands for **A**ttribute-**V**alue, **C**lasslabel, because the values of the dependent attribute are often called **class labels**.) For example, consider the InsuranceInfo relation as shown in Figure 24.9. The AVC set of the root node of the tree for predictor attribute *age* is the result of the following database query:

```
SELECT    R.age, R.highrisk, COUNT (*)
FROM      InsuranceInfo R
GROUP BY  R.age, R.highrisk
```

The AVC set for the left child of the root node for predictor attribute *cartype* is the result of the following query:

```
SELECT    R.cartype, R.highrisk, COUNT (*)
FROM      InsuranceInfo I
```

```
WHERE     R.age <= 25
GROUP BY R.age, R.highrisk
```

The two AVC sets of the root node of the tree are shown in Figure 24.10.

Car type	highrisk	
	true	false
Sedan	0	4
Sports	1	1
Truck	2	1

Age	highrisk	
	true	false
18	0	1
23	1	1
25	2	0
30	0	3
36	0	1

Figure 24.10 AVC Group of the Root Node for the InsuranceInfo Relation

We define the **AVC group** of a node n to be the set of the AVC sets of all predictor attributes at node n. In our example of the InsuranceInfo relation, there are two predictor attributes; therefore, the AVC group of any node consists of two AVC sets.

How large are AVC sets? Note that the size of the AVC set of a predictor attribute X at node n depends only on the number of distinct attribute values of X and the size of the domain of the dependent attribute. For example, consider the AVC sets shown in Figure 24.10. The AVC set for the predictor attribute *cartype* has three entries, and the AVC set for predictor attribute *age* has five entries, although the InsuranceInfo relation as shown in Figure 24.9 has nine records. For large databases, the size of the AVC sets is independent of the number of tuples in the database, except if there are attributes with very large domains, e.g., a real-valued field that is recorded at a very high precision with many digits after the decimal point.

If we make the simplifying assumption that all the AVC sets of the root node together fit into main memory, then we can construct decision trees from very large databases as follows: We make a scan over the database and construct the AVC group of the root node in memory. Then we run the split selection method of our choice with the AVC group as input. After the split selection method computes the splitting attribute and the splitting predicates on the outgoing nodes, we partition the database and recurse. Note that this algorithm is very similar to the original algorithm shown in Figure 24.8; the only modification necessary is shown in Figure 24.11. In addition, this algorithm is still independent of the actual split selection method involved.

24.5 CLUSTERING

In this section we discuss the **clustering problem**. The goal is to partition a set of records into groups such that records within a group are similar to each other and

Input: node n, partition D, split selection method $\mathcal{S}$
Output: decision tree for D rooted at node n

Top-Down Decision Tree Induction Schema:
BuildTree(Node n, data partition D, split selection method $\mathcal{S}$)
(1a) Make a scan over D and construct the AVC group of n in-memory
(1b) Apply $\mathcal{S}$ to the AVC group to find the splitting criterion

Figure 24.11 Classification Tree Induction Refinement with AVC Groups

records that belong to two different groups are dissimilar. Each such group is called a **cluster** and each record belongs to exactly one cluster.[1] Similarity between records is measured computationally by a **distance function**. A distance function takes two input records and returns a value that is a measure of their similarity. Different applications have different notions of similarity and there is no one measure that works for all domains.

As an example, consider the schema of the CustomerInfo view:

CustomerInfo(*age:* `int`, *salary:* `real`)

We can plot the records in the view on a two-dimensional plane as shown in Figure 24.12. The two coordinates of a record are the values of the record's *salary* and *age* fields. We can visually identify three clusters: Young customers who have low salaries, young customers with high salaries, and older customers with high salaries.

Usually, the output of a clustering algorithm consists of a **summarized representation** of each cluster. The type of summarized representation depends strongly on the type and shape of clusters the algorithm computes. For example, assume that we have spherical clusters as in the example shown in Figure 24.12. We can summarize each cluster by its *center* (often also called the *mean*) and its *radius* which are defined as follows. Given a collection of records $r_1, \ldots, r_n$, their **center** C and **radius** R are defined as follows:

$$C = \sum_{i=1}^{n} r_i, \text{ and } R = \sqrt{\frac{\sum_{i=1}^{n}(r_i - M)}{n}}$$

There are two types of clustering algorithms. A **partitional** clustering algorithm partitions the data into k groups such that some criterion that evaluates the clustering quality is optimized. The number of clusters k is a parameter whose value is specified

[1] There are clustering algorithms that allow overlapping clusters, where a record can potentially belong to several clusters.

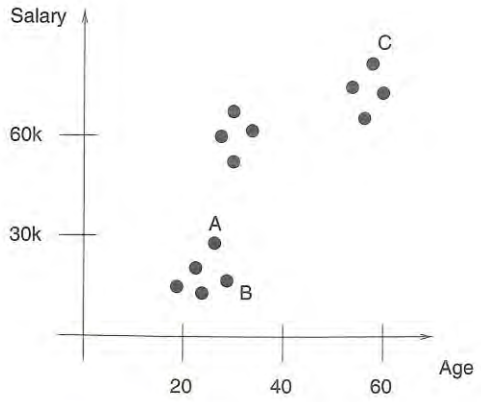


Figure 24.12 Records in CustomerInfo

by the user. A **hierarchical** clustering algorithm generates a sequence of partitions of the records. Starting with a partition in which each cluster consists of one single record, the algorithm merges two partitions in each step until only one single partition remains in the end.

24.5.1 A Clustering Algorithm

Clustering is a very old problem and numerous algorithms have been developed to cluster a collection of records. Traditionally, the number of records in the input database was assumed to be relatively small and the complete database was assumed to fit into main memory. In this section we describe a clustering algorithm called BIRCH that handles very large databases. The design of BIRCH reflects the following two assumptions:

- The number of records is potentially very large and therefore we want to make only one scan over the database.
- We have only a limited amount of main memory available.

A user can set two parameters to control the BIRCH algorithm. The first parameter is a threshold on the amount of main memory available. This main memory threshold translates into a maximum number of cluster summaries k that can be maintained in memory. The second parameter ϵ is an initial threshold for the radius of any cluster. The value of ϵ is an upper bound on the radius of any cluster and controls the number of clusters that the algorithm discovers. If ϵ is small, we discover many small clusters; if ϵ is large, we discover very few clusters, each of which is relatively large. We say that a cluster is **compact** if its radius is smaller than ϵ.

BIRCH always maintains k or fewer cluster summaries (C_i, R_i) in main memory, where C_i is the center of cluster i and R_i is the radius of cluster i. The algorithm always maintains compact clusters, i.e., the radius of each cluster is less than ϵ. If this invariant cannot be maintained with the given amount of main memory, ϵ is increased as described below.

The algorithm reads records from the database sequentially and processes them as follows:

1. Compute the distance between record r and each of the existing cluster centers. Let i be the cluster index such that the distance between r and C_i is the smallest.
2. Compute the value of the new radius R_i' of the ith cluster under the assumption that r is inserted into it. If $R_i' \leq \epsilon$, then the ith cluster remains compact and we assign r to the ith cluster by updating its center and setting its radius to R_i'. If $R_i' > \epsilon$, then the ith cluster is no longer compact if we insert r into it. Therefore, we start a new cluster containing only the record r.

The second step above presents a problem if we already have the maximum number of cluster summaries, k. If we now read a record that requires us to create a new cluster, we don't have the main memory required to hold its summary. In this case, we increase the radius threshold ϵ—using some heuristic to determine the increase—in order to *merge* existing clusters: An increase of ϵ has two consequences. First, existing clusters can accommodate 'more' records, since their maximum radius has increased. Second, it might be possible to merge existing clusters such that the resulting cluster is still compact. Thus, an increase in ϵ usually reduces the number of existing clusters.

The complete BIRCH algorithm uses a balanced in-memory tree, which is similar to a B+ tree in structure, to quickly identify the closest cluster center for a new record. A description of this data structure is beyond the scope of our discussion.

24.6 SIMILARITY SEARCH OVER SEQUENCES

A lot of information stored in databases consists of sequences. In this section, we introduce the problem of similarity search over a collection of sequences. Our query model is very simple: We assume that the user specifies a **query sequence** and wants to retrieve all data sequences that are similar to the query sequence. Similarity search is different from 'normal' queries in that we are not only interested in sequences that match the query sequence exactly, but also in sequences that differ only slightly from the query sequence.

We begin by describing sequences and similarity between sequences. A **data sequence** X is a series of numbers $X = \langle x_1, \ldots, x_k \rangle$. Sometimes X is also called a **time series**. We call k the **length** of the sequence. A **subsequence** $Z = \langle z_1, \ldots, z_j \rangle$ is obtained

from another sequence $X = \langle x_1, \ldots, x_k \rangle$ by deleting numbers from the front and back of the sequence X. Formally, Z is a subsequence of X if $z_1 = x_i, z_2 = x_{i+1}, \ldots, z_j = z_{i+j-1}$ for some $i \in \{1, \ldots, k - j + 1\}$. Given two sequences $X = \langle x_1, \ldots, x_k \rangle$ and $Y = \langle y_1, \ldots, y_k \rangle$, we can define the **Euclidean norm** as the distance between the two sequences as follows:

$$\|X - Y\| = \sum_{i=1}^{k} (x_i - y_i)^2$$

Given a user-specified query sequence and a threshold parameter ϵ, our goal is to retrieve all data sequences that are within ϵ-distance to the query sequence.

Similarity queries over sequences can be classified into two types.

- **Complete sequence matching:** The query sequence and the sequences in the database have the same length. Given a user-specified threshold parameter ϵ, our goal is to retrieve all sequences in the database that are within ϵ-distance to the query sequence.
- **Subsequence matching:** The query sequence is shorter than the sequences in the database. In this case, we want to find all subsequences of sequences in the database such that the subsequence is within distance ϵ of the query sequence. We will not discuss subsequence matching.

24.6.1 An Algorithm to Find Similar Sequences

Given a collection of data sequences, a query sequence, and a distance threshold ϵ, how can we efficiently find all sequences that are within ϵ-distance from the query sequence?

One possibility is to scan the database, retrieve each data sequence, and compute its distance to the query sequence. Even though this algorithm is very simple, it always retrieves every data sequence.

Because we consider the complete sequence matching problem, all data sequences and the query sequence have the same length. We can think of this similarity search as a high-dimensional indexing problem. Each data sequence and the query sequence can be represented as a point in a k-dimensional space. Thus, if we insert all data sequences into a multidimensional index, we can retrieve data sequences that exactly match the query sequence by querying the index. But since we want to retrieve not only data sequences that match the query exactly, but also all sequences that are within ϵ-distance from the query sequence, we do not use a point query as defined by the query sequence. Instead, we query the index with a hyper-rectangle that has side-length $2 \cdot \epsilon$ and the query sequence as center, and we retrieve all sequences that

Two example data mining products—IBM Intelligent Miner and Silicon Graphics Mineset: Both products offer a wide range of data mining algorithms including association rules, regression, classification, and clustering. The emphasis of Intelligent Miner is on scalability—the product contains versions of all algorithms for parallel computers and is tightly integrated with IBM's DB2 database system. Mineset supports extensive visualization of all data mining results, utilizing the powerful graphics features of SGI workstations.

fall within this hyper-rectangle. We then discard sequences that are actually further than only a distance of ϵ away from the query sequence.

Using the index allows us to greatly reduce the number of sequences that we consider and decreases the time to evaluate the similarity query significantly. The references at the end of the chapter provide pointers to further improvements.

24.7 ADDITIONAL DATA MINING TASKS

We have concentrated on the problem of discovering patterns from a database. There are several other equally important data mining tasks, some of which we discuss briefly below. The bibliographic references at the end of the chapter provide many pointers for further study.

Dataset and feature selection: It is often important to select the 'right' dataset to mine. Dataset selection is the process of finding which datasets to mine. Feature selection is the process of deciding which attributes to include in the mining process.

Sampling: One way to explore a large dataset is to obtain one or more *samples* and to analyze the samples. The advantage of sampling is that we can carry out detailed analysis on a sample that would be infeasible on the entire dataset, for very large datasets. The disadvantage of sampling is that obtaining a representative sample for a given task is difficult; we might miss important trends or patterns because they are not reflected in the sample. Current database systems also provide poor support for efficiently obtaining samples. Improving database support for obtaining samples with various desirable statistical properties is relatively straightforward and is likely to be available in future DBMSs. Applying sampling for data mining is an area for further research.

Visualization: Visualization techniques can significantly assist in understanding complex datasets and detecting interesting patterns, and the importance of visualization in data mining is widely recognized.

24.8 POINTS TO REVIEW

- *Data mining* consists of finding interesting patterns in large datasets. It is part of an iterative process that involves data source selection, preprocessing, transformation, data mining, and finally interpretation of results. **(Section 24.1)**
- An *itemset* is a collection of items purchased by a customer in a single *customer transaction.* Given a database of transactions, we call an itemset *frequent* if it is contained in a user-specified percentage of all transactions. The *a priori property* is that every subset of a frequent itemset is also frequent. We can identify frequent itemsets efficiently through a bottom-up algorithm that first generates all frequent itemsets of size one, then size two, and so on. We can prune the search space of candidate itemsets using the a priori property. *Iceberg queries* are `SELECT-FROM-GROUP BY-HAVING` queries with a condition involving aggregation in the `HAVING` clause. Iceberg queries are amenable to the same bottom-up strategy that is used for computing frequent itemsets. **(Section 24.2)**
- An important type of pattern that we can discover from a database is a rule. *Association rules* have the form LHS $\Rightarrow$ RHS with the interpretation that if every item in the LHS is purchased, then it is likely that items in the RHS are purchased as well. Two important measures for a rule are its *support* and *confidence.* We can compute all association rules with user-specified support and confidence thresholds by post-processing frequent itemsets. Generalizations of association rules involve an *ISA hierarchy* on the items and more general grouping conditions that extend beyond the concept of a customer transaction. A *sequential pattern* is a sequence of itemsets purchased by the same customer. The type of rules that we discussed describe associations in the database and do not imply causal relationships. *Bayesian networks* are graphical models that can represent causal relationships. *Classification* and *regression rules* are more general rules that involve *numerical* and *categorical* attributes. **(Section 24.3)**
- Classification and regression rules are often represented in the form of a tree. If a tree represents a collection of classification rules, it is often called a *decision tree.* Decision trees are constructed greedily top-down. A *split selection method* selects the *splitting criterion* at each node of the tree. A relatively compact data structure, the *AVC set* contains sufficient information to let split selection methods decide on the splitting criterion. **(Section 24.4)**
- *Clustering* aims to partition a collection of records into groups called *clusters* such that similar records fall into the same cluster and dissimilar records fall into different clusters. Similarity is usually based on a *distance function.* **(Section 24.5)**
- *Similarity queries* are different from exact queries in that we also want to retrieve results that are slightly different from the exact answer. A *sequence* is an ordered series of numbers. We can measure the difference between two sequences by computing the *Euclidean distance* between the sequences. In *similarity search*

over sequences, we are given a collection of *data sequences*, a *query sequence*, and a threshold parameter ϵ and want to retrieve all data sequences that are within ϵ-distance from the query sequence. One approach is to represent each sequence as a point in a multidimensional space and then use a multidimensional indexing method to limit the number of candidate sequences returned. **(Section 24.6)**

- Additional data mining tasks include *dataset and feature selection*, *sampling*, and *visualization*. **(Section 24.7)**

EXERCISES

Exercise 24.1 Briefly answer the following questions.

1. Define *support* and *confidence* for an association rule.
2. Explain why association rules cannot be used directly for prediction, without further analysis or domain knowledge.
3. Distinguish between *association rules*, *classification rules*, and *regression rules*.
4. Distinguish between *classification* and *clustering*.
5. What is the role of information visualization in data mining?
6. Give examples of queries over a database of stock price quotes, stored as sequences, one per stock, that cannot be expressed in SQL.

Exercise 24.2 Consider the Purchases table shown in Figure 24.1.

1. Simulate the algorithm for finding frequent itemsets on this table with *minsup*=90 percent, and then find association rules with *minconf*=90 percent.
2. Can you modify the table so that the same frequent itemsets are obtained with *minsup*=90 percent as with *minsup*=70 percent on the table shown in Figure 24.1?
3. Simulate the algorithm for finding frequent itemsets on the table in Figure 24.1 with *minsup*=10 percent and then find association rules with *minconf*=90 percent.
4. Can you modify the table so that the same frequent itemsets are obtained with *minsup*=10 percent as with *minsup*=70 percent on the table shown in Figure 24.1?

Exercise 24.3 Consider the Purchases table shown in Figure 24.1. Find all (generalized) association rules that indicate likelihood of items being purchased on the same date by the same customer, with *minsup*=10 percent and *minconf*=70 percent.

Exercise 24.4 Let us develop a new algorithm for the computation of all large itemsets. Assume that we are given a relation D similar to the Purchases table shown in Figure 24.1. We partition the table horizontally into k parts $D_1, \ldots, D_k$.

1. Show that if itemset x is frequent in D, then it is frequent in at least one of the k parts.
2. Use this observation to develop an algorithm that computes all frequent itemsets in two scans over D. (Hint: In the first scan, compute the locally frequent itemsets for each part D_i, $i \in \{1, \ldots, k\}$.)

3. Illustrate your algorithm using the Purchases table shown in Figure 24.1. The first partition consists of the two transactions with *transid* 111 and 112, the second partition consists of the two transactions with *transid* 113 and 114. Assume that the minimum support is 70 percent.

Exercise 24.5 Consider the Purchases table shown in Figure 24.1. Find all sequential patterns with *minsup*= 60 percent. (The text only sketches the algorithm for discovering sequential patterns; so use brute force or read one of the references for a complete algorithm.)

age	*salary*	*subscription*
37	45k	No
39	70k	Yes
56	50k	Yes
52	43k	Yes
35	90k	Yes
32	54k	No
40	58k	No
55	85k	Yes
43	68k	Yes

Figure 24.13 The SubscriberInfo Relation

Exercise 24.6 Consider the SubscriberInfo Relation shown in Figure 24.13. It contains information about the marketing campaign of the *DB Aficionado* magazine. The first two columns show the age and salary of a potential customer and the *subscription* column shows whether the person subscribed to the magazine. We want to use this data to construct a decision tree that helps to predict whether a person is going to subscribe to the magazine.

1. Construct the AVC-group of the root node of the tree.
2. Assume that the spliting predicate at the root node is $age \leq 50$. Construct the AVC-groups of the two children nodes of the root node.

Exercise 24.7 Assume you are given the following set of six records: $\langle 7, 55\rangle$, $\langle 21, 202\rangle$, $\langle 25, 220\rangle$, $\langle 12, 73\rangle$, $\langle 8, 61\rangle$, and $\langle 22, 249\rangle$.

1. Assuming that all six records belong to a single cluster, compute its center and radius.
2. Assume that the first three records belong to one cluster and the second three records belong to a different cluster. Compute the center and radius of the two clusters.
3. Which of the two clusterings is 'better' in your opinion and why?

Exercise 24.8 Assume you are given the three sequences $\langle 1, 3, 4\rangle$, $\langle 2, 3, 2\rangle$, $\langle 3, 3, 7\rangle$. Compute the Euclidian norm between all pairs of sequences.

BIBLIOGRAPHIC NOTES

Discovering useful knowledge from a large database is more than just applying a collection of data mining algorithms, and the point of view that it is an iterative process guided by

an analyst is stressed in [227] and [579]. Work on exploratory data analysis in statistics, for example, [654], and on machine learning and knowledge discovery in artificial intelligence was a precursor to the current focus on data mining; the added emphasis on large volumes of data is the important new element. Good recent surveys of data mining algorithms include [336, 229, 441]. [228] contains additional surveys and articles on many aspects of data mining and knowledge discovery, including a tutorial on Bayesian networks [313]. The book by Piatetsky-Shapiro and Frawley [518] and the book by Fayyad, Piatetsky-Shapiro, Smyth, and Uthurusamy [230] contain collections of data mining papers. The annual SIGKDD conference, run by the ACM special interest group in knowledge discovery in databases, is a good resource for readers interested in current research in data mining [231, 602, 314, 21], as is the *Journal of Knowledge Discovery and Data Mining.*

The problem of mining association rules was introduced by Agrawal, Imielinski, and Swami [16]. Many efficient algorithms have been proposed for the computation of large itemsets, including [17]. Iceberg queries have been introduced by Fang et al. [226]. There is also a large body of research on generalized forms of association rules; for example [611, 612, 614]. A fast algorithm based on sampling is proposed in [647]. Parallel algorithms are described in [19] and [570]. [249] presents an algorithm for discovering association rules over a continuous numeric attribute; association rules over numeric attributes are also discussed in [687]. The general form of association rules in which attributes other than the transaction id are grouped is developed in [459]. Association rules over items in a hierarchy are discussed in [611, 306]. Further extensions and generalization of association rules are proposed in [98, 492, 352]. Integration of mining for frequent itemsets into database systems has been addressed in [569, 652]. The problem of mining sequential patterns is discussed in [20], and further algorithms for mining sequential patterns can be found in [444, 613].

General introductions to classification and regression rules can be found in [307, 462]. The classic reference for decision and regression tree construction is the CART book by Breiman, Friedman, Olshen, and Stone [94]. A machine learning perspective of decision tree construction is given by Quinlan [526]. Recently, several scalable algorithms for decision tree construction have been developed [264, 265, 453, 539, 587].

The clustering problem has been studied for decades in several disciplines. Sample textbooks include [195, 346, 357]. Sample scalable clustering algorithms include CLARANS [491], DBSCAN [211, 212], BIRCH [698], and CURE [292]. Bradley, Fayyad and Reina address the problem of scaling the K-Means clustering algorithm to large databases [92, 91]. The problem of finding clusters in subsets of the fields is addressed in [15]. Ganti et al. examine the problem of clustering data in arbitrary metric spaces [258]. Algorithms for clustering caterogical data include STIRR [267] and CACTUS [257].

Sequence queries have received a lot of attention recently. Extending relational systems, which deal with sets of records, to deal with sequences of records is investigated in [410, 578, 584]. Finding similar sequences from a large database of sequences is discussed in [18, 224, 385, 528, 592].



Ramakrishnan
Gehrke

Database Management Sys

Second Edition

T1-AQK-423

736

Database Management Systems (*DBMS*) is a must for any course in database systems or file organization. *DBMS* provides a hands-on approach to relational database systems, with an emphasis on practical topics such as indexing methods, SQL, and database design. New to this edition are the early coverage of the ER model, new chapters on Internet databases, data mining, and spatial databases, and a new supplement on practical SQL assignments (with solutions for instructors' use). Many other chapters have been reorganized or expanded to provide up-to-date coverage.

DBMS has an example-driven presentation style that motivates new concepts and is easy to read. The quantitative, hands-on approach helps students to evaluate alternatives and is reinforced through a very extensive collection of exercises. Features of *DBMS* include the following:

- Thorough discussion of SQL based on the SQL-92 and SQL:1999 standards.
- Coverage of all aspects of database design including conceptual design, normalization, physical design, and tuning.
- A comprehensive treatment of file organization and indexing.
- Emphasis on widely used concepts, extensive examples, and detailed chapter reviews.
- In-depth coverage of new topics such as data mining, decision support (including materialized views, data warehouses, and OLAP), spatial databases, and Internet databases (including coverage of XML and search engines).

DBMS has its own web site at **http://www.cs.wisc.edu/~dbbook**. Lecture slides (in MS Powerpoint, Postscript, and PDF formats), exercise solutions, additional material for SQL exercises using MS Access, etc., and the Minibase software can be obtained from this site.

RAGHU RAMAKRISHNAN is a professor of computer science at the University of Wisconsin-Madison, where he has been on the faculty since 1987.

JOHANNES GEHRKE is an assistant professor of computer science at Cornell University.

ISBN 0-07-232206-3

9 780072 322064 90000

www.mhhe.com

McGraw-Hill Higher Education

A Division of The McGraw-Hill Companies