Database Management Systems

Second Edition



Raghu Ramakrishnan / Johannes Gehrke

FreeWheel, Ex. 1056
FreeWheel v. AlmondNet
IPR2024-00413

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

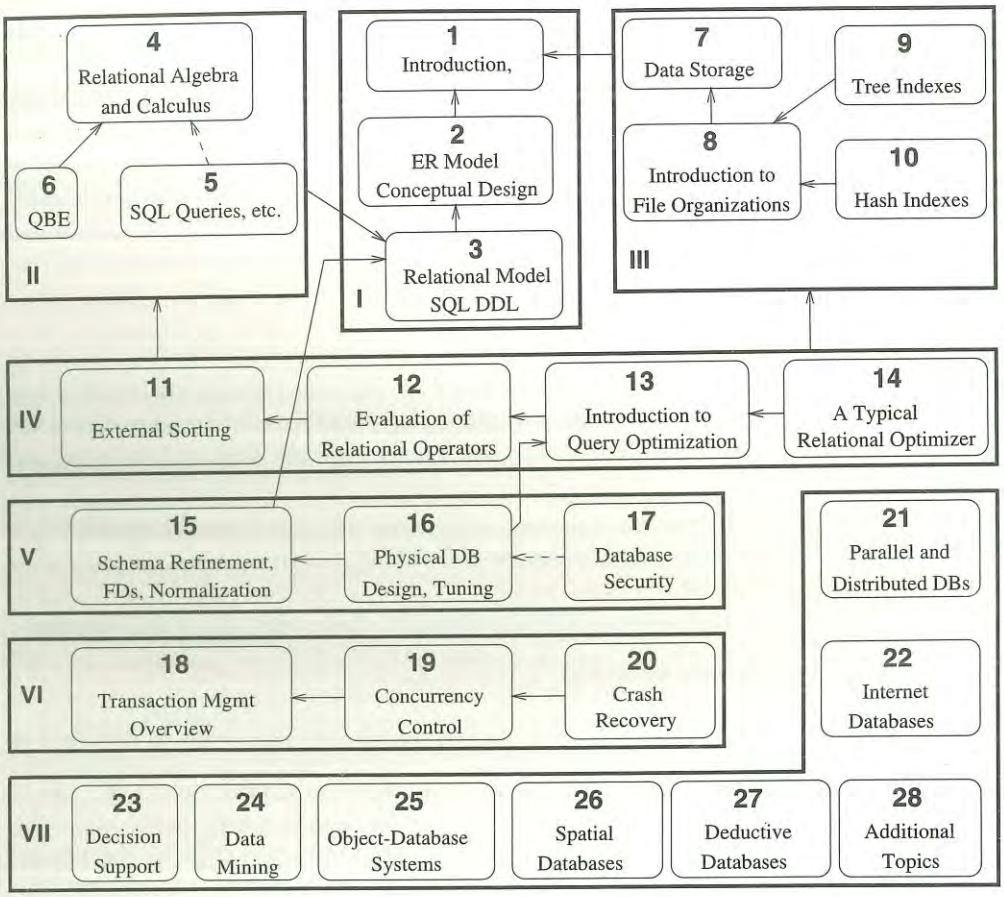


Figure 0.1 Chapter Organization and Dependencies

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.

20

CRASH RECOVERY

Humpty Dumpty sat on a wall.
Humpty Dumpty had a great fall.
All the King's horses and all the King's men
Could not put Humpty together again.

—Old nursery rhyme

The **recovery manager** of a DBMS is responsible for ensuring two important properties of transactions: *atomicity* and *durability*. It ensures atomicity by undoing the actions of transactions that do not commit and durability by making sure that all actions of committed transactions survive **system crashes**, (e.g., a core dump caused by a bus error) and **media failures** (e.g., a disk is corrupted).

The recovery manager is one of the hardest components of a DBMS to design and implement. It must deal with a wide variety of database states because it is called on during system failures. In this chapter, we present the **ARIES** recovery algorithm, which is conceptually simple, works well with a wide range of concurrency control mechanisms, and is being used in an increasing number of database sytems.

We begin with an introduction to ARIES in Section 20.1. We discuss recovery from a crash in Section 20.2. Aborting (or rolling back) a single transaction is a special case of Undo and is discussed in Section 20.2.3. We concentrate on recovery from system crashes in most of the chapter and discuss media failures in Section 20.3. We consider recovery only in a centralized DBMS; recovery in a distributed DBMS is discussed in Chapter 21.

20.1 INTRODUCTION TO ARIES

ARIES is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases:

1. **Analysis:** Identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash.

2. **Redo:** Repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash.

3. **Undo:** Undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions.

Consider the simple execution history illustrated in Figure 20.1. When the system is

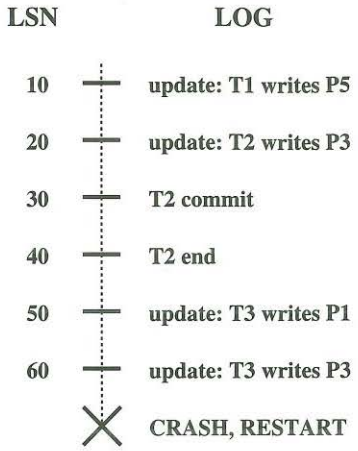


Figure 20.1 Execution History with a Crash

restarted, the Analysis phase identifies $T1$ and $T3$ as transactions that were active at the time of the crash, and therefore to be undone; $T2$ as a committed transaction, and all its actions, therefore, to be written to disk; and $P1$, $P3$, and $P5$ as potentially dirty pages. All the updates (including those of $T1$ and $T3$) are reapplied in the order shown during the Redo phase. Finally, the actions of $T1$ and $T3$ are undone in reverse order during the Undo phase; that is, $T3$'s write of $P3$ is undone, $T3$'s write of $P1$ is undone, and then $T1$'s write of $P5$ is undone.

There are three main principles behind the ARIES recovery algorithm:

- **Write-ahead logging:** Any change to a database object is first recorded in the log; the record in the log must be written to stable storage before the change to the database object is written to disk.

- **Repeating history during Redo:** Upon restart following a crash, ARIES retraces all actions of the DBMS before the crash and brings the system back to the exact state that it was in at the time of the crash. Then, it undoes the actions of transactions that were still active at the time of the crash (effectively aborting them).

Crash recovery: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all use a WAL scheme for recovery. IBM DB2 uses ARIES, and the others use schemes that are actually quite similar to ARIES (e.g., all changes are re-applied, not just the changes made by transactions that are "winners") although there are several variations.

- **Logging changes during Undo:** Changes made to the database while undoing a transaction are logged in order to ensure that such an action is not repeated in the event of repeated (failures causing) restarts.

The second point distinguishes ARIES from other recovery algorithms and is the basis for much of its simplicity and flexibility. In particular, ARIES can support concurrency control protocols that involve locks of finer granularity than a page (e.g., record-level locks). The second and third points are also important in dealing with operations such that redoing and undoing the operation are not exact inverses of each other. We discuss the interaction between concurrency control and crash recovery in Section 20.4, where we also discuss other approaches to recovery briefly.

20.1.1 The Log

The log, sometimes called the **trail** or **journal**, is a history of actions executed by the DBMS. Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes; this durability can be achieved by maintaining two or more copies of the log on different disks (perhaps in different locations), so that the chance of all copies of the log being simultaneously lost is negligibly small.

The most recent portion of the log, called the **log tail**, is kept in main memory and is periodically forced to stable storage. This way, log records and data records are written to disk at the same granularity (pages or sets of pages).

Every **log record** is given a unique *id* called the **log sequence number (LSN)**. As with any record id, we can fetch a log record with one disk access given the LSN. Further, LSNs should be assigned in monotonically increasing order; this property is required for the ARIES recovery algorithm. If the log is a sequential file, in principle growing indefinitely, the LSN can simply be the address of the first byte of the log record.[1]

[1] In practice, various techniques are used to identify portions of the log that are 'too old' to ever be needed again, in order to bound the amount of stable storage used for the log. Given such a bound, the log may be implemented as a 'circular' file, in which case the LSN may be the log record id plus a *wrap-count*.

For recovery purposes, every page in the database contains the LSN of the most recent log record that describes a change to this page. This LSN is called the **pageLSN**.

A log record is written for each of the following actions:

- **Updating a page**: After modifying the page, an *update* type record (described later in this section) is appended to the log tail. The pageLSN of the page is then set to the LSN of the update log record. (The page must be pinned in the buffer pool while these actions are carried out.)
- **Commit**: When a transaction decides to commit, it **force-writes** a *commit* type log record containing the transaction id. That is, the log record is appended to the log, and the log tail is written to stable storage, up to and including the commit record.[2] The transaction is considered to have committed at the instant that its commit log record is written to stable storage. (Some additional steps must be taken, e.g., removing the transaction's entry in the transaction table; these follow the writing of the commit log record.)
- **Abort**: When a transaction is aborted, an *abort* type log record containing the transaction id is appended to the log, and Undo is initiated for this transaction (Section 20.2.3).
- **End**: As noted above, when a transaction is aborted or committed, some additional actions must be taken beyond writing the abort or commit log record. After all these additional steps are completed, an *end* type log record containing the transaction id is appended to the log.
- **Undoing an update:** When a transaction is rolled back (because the transaction is aborted, or during recovery from a crash), its updates are undone. When the action described by an update log record is undone, a *compensation log record*, or CLR, is written.

Every log record has certain fields: **prevLSN**, **transID**, and **type**. The set of all log records for a given transaction is maintained as a linked list going back in time, using the **prevLSN** field; this list must be updated whenever a log record is added. The transID field is the id of the transaction generating the log record, and the type field obviously indicates the type of the log record.

Additional fields depend on the type of the log record. We have already mentioned the additional contents of the various log record types, with the exception of the update and compensation log record types, which we describe next.

[2]Note that this step requires the buffer manager to be able to selectively *force* pages to stable storage.

Update Log Records

The fields in an **update** log record are illustrated in Figure 20.2. The **pageID** field

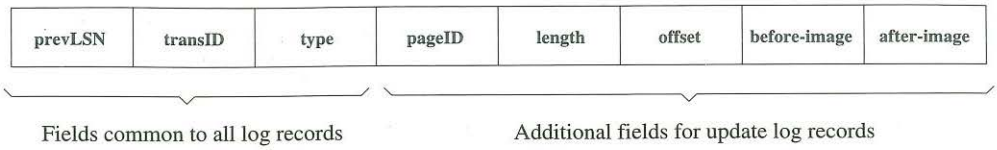


Figure 20.2 Contents of an Update Log Record

is the page id of the modified page; the length in bytes and the offset of the change are also included. The **before-image** is the value of the changed bytes before the change; the **after-image** is the value after the change. An update log record that contains both before- and after-images can be used to redo the change and to undo it. In certain contexts, which we will not discuss further, we can recognize that the change will never be undone (or, perhaps, redone). A **redo-only update** log record will contain just the after-image; similarly an **undo-only update** record will contain just the before-image.

Compensation Log Records

A **compensation log record (CLR)** is written just before the change recorded in an update log record U is undone. (Such an undo can happen during normal system execution when a transaction is aborted or during recovery from a crash.) A compensation log record C describes the action taken to undo the actions recorded in the corresponding update log record and is appended to the log tail just like any other log record. The compensation log record C also contains a field called **undoNextLSN**, which is the LSN of the next log record that is to be undone for the transaction that wrote update record U; this field in C is set to the value of prevLSN in U.

As an example, consider the fourth update log record shown in Figure 20.3. If this update is undone, a CLR would be written, and the information in it would include the transID, pageID, length, offset, and before-image fields from the update record. Notice that the CLR records the (undo) action of changing the affected bytes back to the before-image value; thus, this value and the location of the affected bytes constitute the redo information for the action described by the CLR. The undoNextLSN field is set to the LSN of the first log record in Figure 20.3.

Unlike an update log record, a CLR describes an action that will never be *undone*, that is, we never undo an undo action. The reason is simple: an update log record describes a change made by a transaction during normal execution and the transaction may subsequently be aborted, whereas a CLR describes an action taken to rollback a

transaction for which the decision to abort has already been made. Thus, the transaction *must* be rolled back, and the undo action described by the CLR is definitely required. This observation is very useful because it bounds the amount of space needed for the log during restart from a crash: The number of CLRs that can be written during Undo is no more than the number of update log records for active transactions at the time of the crash.

It may well happen that a CLR is written to stable storage (following WAL, of course) but that the undo action that it describes is not yet written to disk when the system crashes again. In this case the undo action described in the CLR is reapplied during the Redo phase, just like the action described in update log records.

For these reasons, a CLR contains the information needed to reapply, or redo, the change described but not to reverse it.

20.1.2 Other Recovery-Related Data Structures

In addition to the log, the following two tables contain important recovery-related information:

- **Transaction table:** This table contains one entry for each active transaction. The entry contains (among other things) the transaction id, the status, and a field called **lastLSN**, which is the LSN of the most recent log record for this transaction. The **status** of a transaction can be that it is in progress, is committed, or is aborted. (In the latter two cases, the transaction will be removed from the table once certain 'clean up' steps are completed.)
- **Dirty page table:** This table contains one entry for each dirty page in the buffer pool, that is, each page with changes that are not yet reflected on disk. The entry contains a field **recLSN**, which is the LSN of the first log record that caused the page to become dirty. Note that this LSN identifies the earliest log record that might have to be redone for this page during restart from a crash.

During normal operation, these are maintained by the transaction manager and the buffer manager, respectively, and during restart after a crash, these tables are reconstructed in the Analysis phase of restart.

Consider the following simple example. Transaction $T1000$ changes the value of bytes 21 to 23 on page $P500$ from 'ABC' to 'DEF', transaction $T2000$ changes 'HIJ' to 'KLM' on page $P600$, transaction $T2000$ changes bytes 20 through 22 from 'GDE' to 'QRS' on page $P500$, then transaction $T1000$ changes 'TUV' to 'WXY' on page $P505$. The dirty page table, the transaction table,[3] and the log at this instant are shown in Figure

[3]The status field is not shown in the figure for space reasons; all transactions are in progress.

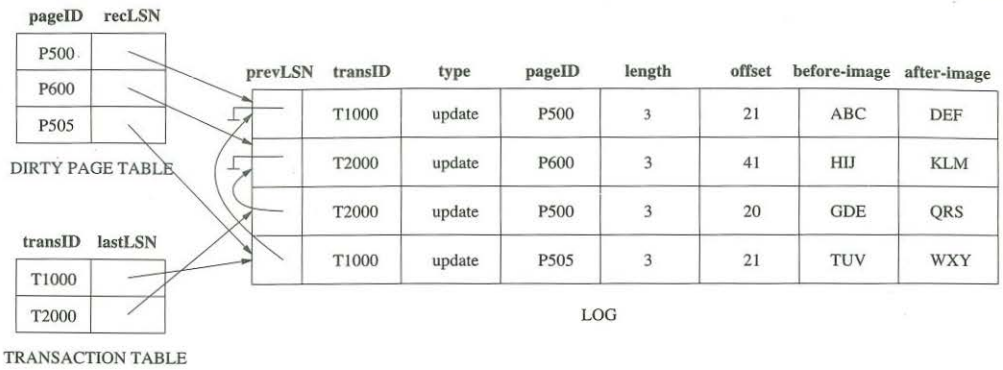


Figure 20.3 Instance of Log and Transaction Table

20.3. Observe that the log is shown growing from top to bottom; older records are at the top. Although the records for each transaction are linked using the prevLSN field, the log as a whole also has a sequential order that is important—for example, $T2000$'s change to page $P500$ follows $T1000$'s change to page $P500$, and in the event of a crash, these changes must be redone in the same order.

20.1.3 The Write-Ahead Log Protocol

Before writing a page to disk, every update log record that describes a change to this page must be forced to stable storage. This is accomplished by forcing all log records up to and including the one with LSN equal to the pageLSN to stable storage before writing the page to disk.

The importance of the WAL protocol cannot be overemphasized—WAL is the fundamental rule that ensures that a record of every change to the database is available while attempting to recover from a crash. If a transaction made a change and committed, the no-force approach means that some of these changes may not have been written to disk at the time of a subsequent crash. Without a record of these changes, there would be no way to ensure that the changes of a committed transaction survive crashes. Note that the definition of a *committed transaction* is effectively "a transaction whose log records, including a commit record, have all been written to stable storage"!

When a transaction is committed, the log tail is forced to stable storage, even if a no-force approach is being used. It is worth contrasting this operation with the actions taken under a force approach: If a force approach is used, all the pages modified by the transaction, rather than a portion of the log that includes all its records, must be forced to disk when the transaction commits. The set of all changed pages is typically much larger than the log tail because the size of an update log record is close to (twice) the size of the changed bytes, which is likely to be much smaller than the page size.

Further, the log is maintained as a sequential file, and thus all writes to the log are sequential writes. Consequently, the cost of forcing the log tail is much smaller than the cost of writing all changed pages to disk.

20.1.4 Checkpointing

A **checkpoint** is like a snapshot of the DBMS state, and by taking checkpoints periodically, as we will see, the DBMS can reduce the amount of work to be done during restart in the event of a subsequent crash.

Checkpointing in ARIES has three steps. First, a **begin_checkpoint** record is written to indicate when the checkpoint starts. Second, an **end_checkpoint** record is constructed, including in it the current contents of the transaction table and the dirty page table, and appended to the log. The third step is carried out after the **end_checkpoint** record is written to stable storage: A special **master** record containing the LSN of the *begin_checkpoint* log record is written to a known place on stable storage. While the end_checkpoint record is being constructed, the DBMS continues executing transactions and writing other log records; the only guarantee we have is that the transaction table and dirty page table are accurate *as of the time of the begin_checkpoint record.*

This kind of checkpoint is called a **fuzzy checkpoint** and is inexpensive because it does not require quiescing the system or writing out pages in the buffer pool (unlike some other forms of checkpointing). On the other hand, the effectiveness of this checkpointing technique is limited by the earliest recLSN of pages in the dirty pages table, because during restart we must redo changes starting from the log record whose LSN is equal to this recLSN. Having a background process that periodically writes dirty pages to disk helps to limit this problem.

When the system comes back up after a crash, the restart process begins by locating the most recent checkpoint record. For uniformity, the system always begins normal execution by taking a checkpoint, in which the transaction table and dirty page table are both empty.

20.2 RECOVERING FROM A SYSTEM CRASH

When the system is restarted after a crash, the recovery manager proceeds in three phases, as shown in Figure 20.4.

The Analysis phase begins by examining the most recent begin_checkpoint record, whose LSN is denoted as C in Figure 20.4, and proceeds forward in the log until the last log record. The Redo phase follows Analysis and redoes all changes to any page that might have been dirty at the time of the crash; this set of pages and the starting

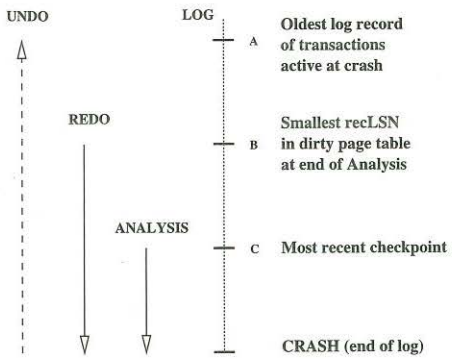


Figure 20.4 Three Phases of Restart in ARIES

point for Redo (the smallest recLSN of any dirty page) are determined during Analysis. The Undo phase follows Redo and undoes the changes of all transactions that were active at the time of the crash; again, this set of transactions is identified during the Analysis phase. Notice that Redo reapplies changes in the order in which they were originally carried out; Undo reverses changes in the opposite order, reversing the most recent change first.

Observe that the relative order of the three points A, B, and C in the log may differ from that shown in Figure 20.4. The three phases of restart are described in more detail in the following sections.

20.2.1 Analysis Phase

The **Analysis** phase performs three tasks:

1. It determines the point in the log at which to start the Redo pass.
2. It determines (a conservative superset of the) pages in the buffer pool that were dirty at the time of the crash.
3. It identifies transactions that were active at the time of the crash and must be undone.

Analysis begins by examining the most recent begin_checkpoint log record and initializing the dirty page table and transaction table to the copies of those structures in the next end_checkpoint record. Thus, these tables are initialized to the set of dirty pages and active transactions at the time of the checkpoint. (If there are additional log records between the begin_checkpoint and end_checkpoint records, the tables must be adjusted to reflect the information in these records, but we omit the details of this

step. See Exercise 20.9.) Analysis then scans the log in the forward direction until it reaches the end of the log:

- If an end log record for a transaction T is encountered, T is removed from the transaction table because it is no longer active.
- If a log record other than an end record for a transaction T is encountered, an entry for T is added to the transaction table if it is not already there. Further, the entry for T is modified:
  1. The lastLSN field is set to the LSN of this log record.
  2. If the log record is a commit record, the status is set to `C`, otherwise it is set to `U` (indicating that it is to be undone).
- If a redoable log record affecting page P is encountered, and P is not in the dirty page table, an entry is inserted into this table with page id P and recLSN equal to the LSN of this redoable log record. This LSN identifies the oldest change affecting page P that may not have been written to disk.

At the end of the Analysis phase, the transaction table contains an accurate list of all transactions that were active at the time of the crash—this is the set of transactions with status `U`. The dirty page table includes all pages that were dirty at the time of the crash, but may also contain some pages that were written to disk. If an *end_write* log record were written at the completion of each write operation, the dirty page table constructed during Analysis could be made more accurate, but in ARIES, the additional cost of writing end_write log records is not considered to be worth the gain.

As an example, consider the execution illustrated in Figure 20.3. Let us extend this execution by assuming that $T2000$ commits, then $T1000$ modifies another page, say, $P700$, and appends an update record to the log tail, and then the system crashes (before this update log record is written to stable storage).

The dirty page table and the transaction table, held in memory, are lost in the crash. The most recent checkpoint is the one that was taken at the beginning of the execution, with an empty transaction table and dirty page table; it is not shown in Figure 20.3. After examining this log record, which we assume is just before the first log record shown in the figure, Analysis initializes the two tables to be empty. Scanning forward in the log, $T1000$ is added to the transaction table; in addition, $P500$ is added to the dirty page table with recLSN equal to the LSN of the first shown log record. Similarly, $T2000$ is added to the transaction table and $P600$ is added to the dirty page table. There is no change based on the third log record, and the fourth record results in the addition of $P505$ to the dirty page table. The commit record for $T2000$ (not in the figure) is now encountered, and $T2000$ is removed from the transaction table.

The Analysis phase is now complete, and it is recognized that the only active transaction at the time of the crash is $T1000$, with lastLSN equal to the LSN of the fourth record in Figure 20.3. The dirty page table reconstructed in the Analysis phase is identical to that shown in the figure. The update log record for the change to $P700$ is lost in the crash and is not seen during the Analysis pass. Thanks to the WAL protocol, however, all is well—the corresponding change to page $P700$ cannot have been written to disk either!

Some of the updates may have been written to disk; for concreteness, let us assume that the change to $P600$ (and only this update) was written to disk before the crash. Thus $P600$ is not dirty, yet it is included in the dirty page table. The pageLSN on page $P600$, however, reflects the write because it is now equal to the LSN of the third update log record shown in Figure 20.3.

20.2.2 Redo Phase

During the **Redo** phase, ARIES reapplies the updates of *all* transactions, committed or otherwise. Further, if a transaction was aborted before the crash and its updates were undone, as indicated by CLRs, the actions described in the CLRs are also reapplied. This **repeating history** paradigm distinguishes ARIES from other proposed WAL-based recovery algorithms and causes the database to be brought to the same state that it was in at the time of the crash.

The Redo phase begins with the log record that has the smallest recLSN of all pages in the dirty page table constructed by the Analysis pass because this log record identifies the oldest update that may not have been written to disk prior to the crash. Starting from this log record, Redo scans forward until the end of the log. For each redoable log record (update or CLR) encountered, Redo checks whether the logged action must be redone. The action must be redone unless one of the following conditions holds:

- The affected page is not in the dirty page table, or
- The affected page is in the dirty page table, but the recLSN for the entry is *greater than* the LSN of the log record being checked, or
- The pageLSN (stored on the page, which must be retrieved to check this condition) is *greater than or equal* to the LSN of the log record being checked.

The first condition obviously means that all changes to this page have been written to disk. Because the recLSN is the first update to this page that may not have been written to disk, the second condition means that the update being checked was indeed propagated to disk. The third condition, which is checked last because it requires us to retrieve the page, also ensures that the update being checked was written to disk,

because either this update or a later update to the page was written. (Recall our assumption that a write to a page is atomic; this assumption is important here!)

If the logged action must be redone:

1. The logged action is reapplied.
2. The pageLSN on the page is set to the LSN of the redone log record. No additional log record is written at this time.

Let us continue with the example discussed in Section 20.2.1. From the dirty page table, the smallest recLSN is seen to be the LSN of the first log record shown in Figure 20.3. Clearly the changes recorded by earlier log records (there happen to be none in this example) have all been written to disk. Now, Redo fetches the affected page, $P500$, and compares the LSN of this log record with the pageLSN on the page and, because we assumed that this page was not written to disk before the crash, finds that the pageLSN is less. The update is therefore reapplied; bytes 21 through 23 are changed to 'DEF', and the pageLSN is set to the LSN of this update log record.

Redo then examines the second log record. Again, the affected page, $P600$, is fetched and the pageLSN is compared to the LSN of the update log record. In this case, because we assumed that $P600$ was written to disk before the crash, they are equal, and the update does not have to be redone.

The remaining log records are processed similarly, bringing the system back to the exact state it was in at the time of the crash. Notice that the first two conditions indicating that a redo is unnecessary never hold in this example. Intuitively, they come into play when the dirty page table contains a very old recLSN, going back to before the most recent checkpoint. In this case, as Redo scans forward from the log record with this LSN, it will encounter log records for pages that were written to disk prior to the checkpoint and were therefore not in the dirty page table in the checkpoint. Some of these pages may be dirtied again after the checkpoint; nonetheless, the updates to these pages prior to the checkpoint need not be redone. Although the third condition alone is sufficient to recognize that these updates need not be redone, it requires us to fetch the affected page. The first two conditions allow us to recognize this situation without fetching the page. (The reader is encouraged to construct examples that illustrate the use of each of these conditions; see Exercise 20.8.)

At the end of the Redo phase, end type records are written for all transactions with status C, which are removed from the transaction table.

20.2.3 Undo Phase

The Undo phase, unlike the other two phases, scans backward from the end of the log. The goal of this phase is to undo the actions of all transactions that were active at the time of the crash, that is, to effectively abort them. This set of transactions is identified in the transaction table constructed by the Analysis phase.

The Undo Algorithm

Undo begins with the transaction table constructed by the Analysis phase, which identifies all transactions that were active at the time of the crash, and includes the LSN of the most recent log record (the lastLSN field) for each such transaction. Such transactions are called **loser transactions**. All actions of losers must be undone, and further, these actions must be undone in the reverse of the order in which they appear in the log.

Consider the set of lastLSN values for all loser transactions. Let us call this set **ToUndo**. Undo repeatedly chooses the largest (i.e., most recent) LSN value in this set and processes it, until ToUndo is empty. To process a log record:

1. If it is a CLR, and the undoNextLSN value is not *null*, the undoNextLSN value is added to the set ToUndo; if the undoNextLSN is *null*, an end record is written for the transaction because it is completely undone, and the CLR is discarded.
2. If it is an update record, a CLR is written and the corresponding action is undone, as described in Section 20.1.1, and the prevLSN value in the update log record is added to the set ToUndo.

When the set ToUndo is empty, the Undo phase is complete. Restart is now complete, and the system can proceed with normal operations.

Let us continue with the scenario discussed in Sections 20.2.1 and 20.2.2. The only active transaction at the time of the crash was determined to be $T1000$. From the transaction table, we get the LSN of its most recent log record, which is the fourth update log record in Figure 20.3. The update is undone, and a CLR is written with undoNextLSN equal to the LSN of the first log record in the figure. The next record to be undone for transaction $T1000$ is the first log record in the figure. After this is undone, a CLR and an end log record for $T1000$ are written, and the Undo phase is complete.

In this example, undoing the action recorded in the first log record causes the action of the third log record, which is due to a committed transaction, to be overwritten and thereby lost! This situation arises because $T2000$ overwrote a data item written by

$T1000$ while $T1000$ was still active; if Strict 2PL were followed, $T2000$ would not have been allowed to overwrite this data item.

Aborting a Transaction

Aborting a transaction is just a special case of the Undo phase of Restart in which a single transaction, rather than a set of transactions, is undone. The example in Figure 20.5, discussed next, illustrates this point.

Crashes during Restart

It is important to understand how the Undo algorithm presented in Section 20.2.3 handles repeated system crashes. Because the details of precisely how the action described in an update log record is undone are straightforward, we will discuss Undo in the presence of system crashes using an execution history, shown in Figure 20.5, that abstracts away unnecessary detail. This example illustrates how aborting a transaction is a special case of Undo and how the use of CLRs ensures that the Undo action for an update log record is not applied twice.

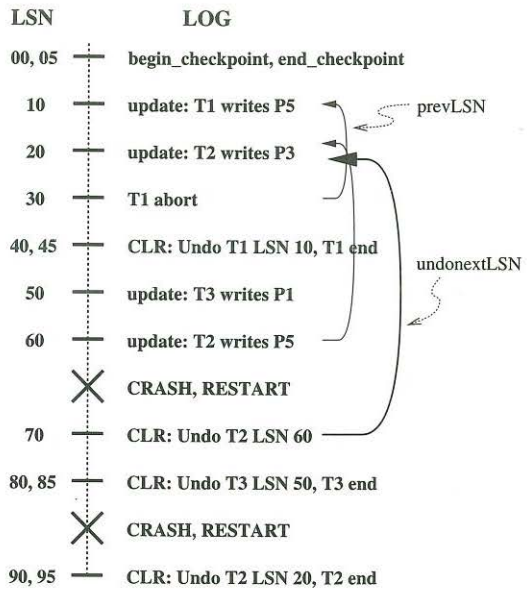


Figure 20.5 Example of Undo with Repeated Crashes

The log shows the order in which the DBMS executed various actions; notice that the LSNs are in ascending order, and that each log record for a transaction has a prevLSN

field that points to the previous log record for that transaction. We have not shown *null* prevLSNs, that is, some special value used in the prevLSN field of the first log record for a transaction to indicate that there is no previous log record. We have also compacted the figure by occasionally displaying two log records (separated by a comma) on a single line.

Log record (with LSN) 30 indicates that $T1$ aborts. All actions of this transaction should be undone in reverse order, and the only action of $T1$, described by the update log record 10, is indeed undone as indicated by CLR 40.

After the first crash, Analysis identifies $P1$ (with recLSN 50), $P3$ (with recLSN 20), and $P5$ (with recLSN 10) as dirty pages. Log record 45 shows that $T1$ is a completed transaction; thus, the transaction table identifies $T2$ (with lastLSN 60) and $T3$ (with lastLSN 50) as active at the time of the crash. The Redo phase begins with log record 10, which is the minimum recLSN in the dirty page table, and reapplies all actions (for the update and CLR records), as per the Redo algorithm presented in Section 20.2.2.

The ToUndo set consists of LSNs 60, for $T2$, and 50, for $T3$. The Undo phase now begins by processing the log record with LSN 60 because 60 is the largest LSN in the ToUndo set. The update is undone, and a CLR (with LSN 70) is written to the log. This CLR has undoNextLSN equal to 20, which is the prevLSN value in log record 60; 20 is the next action to be undone for $T2$. Now the largest remaining LSN in the ToUndo set is 50. The write corresponding to log record 50 is now undone, and a CLR describing the change is written. This CLR has LSN 80, and its undoNextLSN field is *null* because 50 is the only log record for transaction $T3$. Thus $T3$ is completely undone, and an end record is written. Log records 70, 80, and 85 are written to stable storage before the system crashes a second time; however, the changes described by these records may not have been written to disk.

When the system is restarted after the second crash, Analysis determines that the only active transaction at the time of the crash was $T2$; in addition, the dirty page table is identical to what it was during the previous restart. Log records 10 through 85 are processed again during Redo. (If some of the changes made during the previous Redo were written to disk, the pageLSNs on the affected pages are used to detect this situation and avoid writing these pages again.) The Undo phase considers the only LSN in the ToUndo set, 70, and processes it by adding the undoNextLSN value (20) to the ToUndo set. Next, log record 20 is processed by undoing $T2$'s write of page $P3$, and a CLR is written (LSN 90). Because 20 is the first of $T2$'s log records—and therefore, the last of its records to be undone—the undoNextLSN field in this CLR is *null*, an end record is written for $T2$, and the ToUndo set is now empty.

Recovery is now complete, and normal execution can resume with the writing of a checkpoint record.

This example illustrated repeated crashes during the Undo phase. For completeness, let us consider what happens if the system crashes while Restart is in the Analysis or Redo phases. If a crash occurs during the Analysis phase, all the work done in this phase is lost, and on restart the Analysis phase starts afresh with the same information as before. If a crash occurs during the Redo phase, the only effect that survives the crash is that some of the changes made during Redo may have been written to disk prior to the crash. Restart starts again with the Analysis phase and then the Redo phase, and some update log records that were redone the first time around will not be redone a second time because the pageLSN will now be equal to the update record's LSN (although the pages will have to fetched again to detect this).

We can take checkpoints during Restart to minimize repeated work in the event of a crash, but we will not discuss this point.

20.3 MEDIA RECOVERY

Media recovery is based on periodically making a copy of the database. Because copying a large database object such as a file can take a long time, and the DBMS must be allowed to continue with its operations in the meantime, creating a copy is handled in a manner similar to taking a fuzzy checkpoint.

When a database object such as a file or a page is corrupted, the copy of that object is brought up-to-date by using the log to identify and reapply the changes of committed transactions and undo the changes of uncommitted transactions (as of the time of the media recovery operation).

The begin_checkpoint LSN of the most recent complete checkpoint is recorded along with the copy of the database object in order to minimize the work in reapplying changes of committed transactions. Let us compare the smallest recLSN of a dirty page in the corresponding end_checkpoint record with the LSN of the begin_checkpoint record and call the smaller of these two LSNs I. We observe that the actions recorded in all log records with LSNs less than I must be reflected in the copy. Thus, only log records with LSNs greater than I need to be reapplied to the copy.

Finally, the updates of transactions that are incomplete at the time of media recovery or that were aborted after the fuzzy copy was completed need to be undone to ensure that the page reflects only the actions of committed transactions. The set of such transactions can be identified as in the Analysis pass, and we omit the details.

20.4 OTHER ALGORITHMS AND INTERACTION WITH CONCURRENCY CONTROL

Like ARIES, the most popular alternative recovery algorithms also maintain a log of database actions according to the WAL protocol. A major distinction between ARIES and these variants is that the Redo phase in ARIES *repeats history*, that is, redoes the actions of *all* transactions, not just the non-losers. Other algorithms redo only the non-losers, and the Redo phase follows the Undo phase, in which the actions of losers are rolled back.

Thanks to the repeating history paradigm and the use of CLRs, ARIES is able to support fine-granularity locks (record-level locks) and logging of logical operations, rather than just byte-level modifications. For example, consider a transaction T that inserts a data entry 15* into a B+ tree index. Between the time this insert is done and the time that T is eventually aborted, other transactions may also insert and delete entries from the tree. If record-level locks are set, rather than page-level locks, it is possible that the entry 15* is on a different physical page when T aborts from the one that T inserted it into. In this case the undo operation for the insert of 15* must be recorded in logical terms because the physical (byte-level) actions involved in undoing this operation are not the inverse of the physical actions involved in inserting the entry.

Logging logical operations yields considerably higher concurrency, although the use of fine-granularity locks can lead to increased locking activity (because more locks must be set). Thus, there is a trade-off between different WAL-based recovery schemes. We have chosen to cover ARIES because it has several attractive properties, in particular, its simplicity and its ability to support fine-granularity locks and logging of logical operations.

One of the earliest recovery algorithms, used in the System R prototype at IBM, takes a very different approach. There is no logging and, of course, no WAL protocol. Instead, the database is treated as a collection of pages and accessed through a **page table**, which maps page ids to disk addresses. When a transaction makes changes to a data page, it actually makes a copy of the page, called the **shadow** of the page, and changes the shadow page. The transaction copies the appropriate part of the page table and changes the entry for the changed page to point to the shadow, so that it can see the changes; however, other transactions continue to see the original page table, and therefore the original page, until this transaction commits. Aborting a transaction is simple: just discard its shadow versions of the page table and the data pages. Committing a transaction involves making its version of the page table public and discarding the original data pages that are superseded by shadow pages.

This scheme suffers from a number of problems. First, data becomes highly fragmented due to the replacement of pages by shadow versions, which may be located far from

the original page. This phenomenon reduces data clustering and makes good garbage collection imperative. Second, the scheme does not yield a sufficiently high degree of concurrency. Third, there is a substantial storage overhead due to the use of shadow pages. Fourth, the process aborting a transaction can itself run into deadlocks, and this situation must be specially handled because the semantics of aborting an abort transaction gets murky.

For these reasons, even in System R, shadow paging was eventually superseded by WAL-based recovery techniques.

20.5 POINTS TO REVIEW

- ARIES is a very popular recovery algorithm with a steal, no-force approach. Compared to other recovery algorithms, ARIES is relatively simple, and supports locks at a finer granularity than a page. After a crash, restart in ARIES proceeds in three phases. The *Analysis phase* identifies dirty buffer pool pages and active transactions at the time of the crash. The *Redo phase* restores the database to its state at the time of the crash. The *Rndo phase* undoes actions of transactions that did not commit due to the crash. ARIES maintains a *log* of all actions executed by the DBMS. Each log record has a unique *log sequence number (LSN)*. Log records are written for updates, commits, aborts, and whenever an update is undone. In addition, a special *end* record is written whenever the DBMS finishes processing a transaction. ARIES is a *write-ahead log protocol*. Before writing a page to disk or before committing a transaction, every log record that refers to the page or transaction is forced to disk. To reduce the amount of work to be done during a restart, the DBMS takes occasional snapshots called *checkpoints*. The restart process then starts at the most recent checkpoint. **(Section 20.1)**

- After a system crash, the Analysis, Redo, and Undo phases are executed. The redo phase *repeats history* by transforming the database into its state before the crash. The Undo phase undoes actions by *loser transaction*, transactions that are aborted since they were active at the time of the crash. ARIES handles subsequent crashes during system restart by writing *compensating log records (CLRs)* when undoing actions of aborted transaction. CLRs indicate which actions have already been undone and prevent undoing the same action twice. **(Section 20.2)**

- To be able to recover from media failure without reading the complete log, we periodically make a copy of the database. The procedure of copying the database is similar to creating a checkpoint. **(Section 20.3)**

- There are other recovery schemes based on write-ahead logging and there is a recovery scheme that makes changes to copies of data pages, or *shadow pages*. Aries and its variants are replacing these other schemes because of their relative simplic-

ity and ability to support fine-granularity locks and logging of logical operations. **(Section 20.4)**

EXERCISES

Exercise 20.1 Briefly answer the following questions:

1. How does the recovery manager ensure atomicity of transactions? How does it ensure durability?
2. What is the difference between stable storage and disk?
3. What is the difference between a system crash and a media failure?
4. Explain the WAL protocol.
5. Describe the steal and no-force policies.

Exercise 20.2 Briefly answer the following questions:

1. What are the properties required of LSNs?
2. What are the fields in an update log record? Explain the use of each field.
3. What are redoable log records?
4. What are the differences between update log records and CLRs?

Exercise 20.3 Briefly answer the following questions:

1. What are the roles of the Analysis, Redo and Undo phases in ARIES?
2. Consider the execution shown in Figure 20.6.
   (a) What is done during Analysis? (Be precise about the points at which Analysis begins and ends and describe the contents of any tables constructed in this phase.)
   (b) What is done during Redo? (Be precise about the points at which Redo begins and ends.)
   (c) What is done during Undo? (Be precise about the points at which Undo begins and ends.)

Exercise 20.4 Consider the execution shown in Figure 20.7.

1. Extend the figure to show prevLSN and undonextLSN values.
2. Describe the actions taken to rollback transaction $T2$.
3. Show the log after $T2$ is rolled back, including all prevLSN and undonextLSN values in log records.

Exercise 20.5 Consider the execution shown in Figure 20.8. In addition, the system crashes during recovery after writing two log records to stable storage and again after writing another two log records.

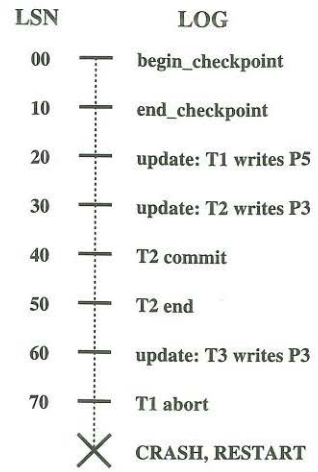


Figure 20.6 Execution with a Crash

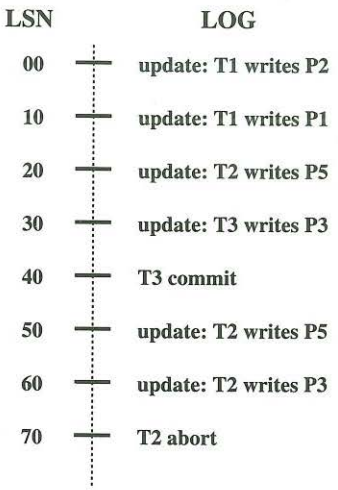


Figure 20.7 Aborting a Transaction

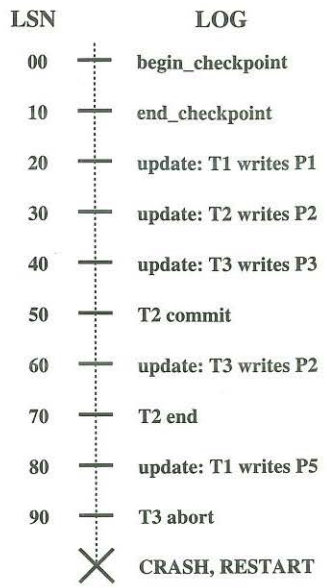


Figure 20.8 Execution with Multiple Crashes

1. What is the value of the LSN stored in the master log record?
2. What is done during Analysis?
3. What is done during Redo?
4. What is done during Undo?
5. Show the log when recovery is complete, including all non-null prevLSN and undonextLSN values in log records.

Exercise 20.6 Briefly answer the following questions:

1. How is checkpointing done in ARIES?
2. Checkpointing can also be done as follows: Quiesce the system so that only checkpointing activity can be in progress, write out copies of all dirty pages, and include the dirty page table and transaction table in the checkpoint record. What are the pros and cons of this approach versus the checkpointing approach of ARIES?
3. What happens if a second begin_checkpoint record is encountered during the Analysis phase?
4. Can a second end_checkpoint record be encountered during the Analysis phase?
5. Why is the use of CLRs important for the use of undo actions that are not the physical inverse of the original update?
6. Give an example that illustrates how the paradigm of repeating history and the use of CLRs allow ARIES to support locks of finer granularity than a page.

Exercise 20.7 Briefly answer the following questions:

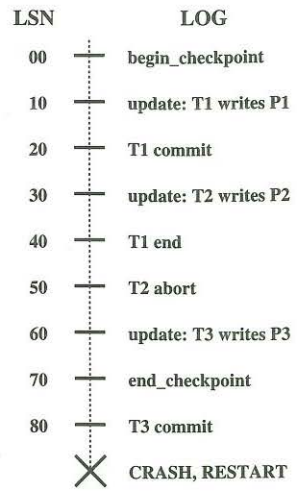


Figure 20.9 Log Records between Checkpoint Records

1. If the system fails repeatedly during recovery, what is the maximum number of log records that can be written (as a function of the number of update and other log records written before the crash) before restart completes successfully?
2. What is the oldest log record that we need to retain?
3. If a bounded amount of stable storage is used for the log, how can we ensure that there is always enough stable storage to hold all log records written during restart?

Exercise 20.8 Consider the three conditions under which a redo is unnecessary (Section 20.2.2).

1. Why is it cheaper to test the first two conditions?
2. Describe an execution that illustrates the use of the first condition.
3. Describe an execution that illustrates the use of the second condition.

Exercise 20.9 The description in Section 20.2.1 of the Analysis phase made the simplifying assumption that no log records appeared between the begin_checkpoint and end_checkpoint records for the most recent complete checkpoint. The following questions explore how such records should be handled.

1. Explain why log records could be written between the begin_checkpoint and end_checkpoint records.
2. Describe how the Analysis phase could be modified to handle such records.
3. Consider the execution shown in Figure 20.9. Show the contents of the end_checkpoint record.
4. Illustrate your modified Analysis phase on the execution shown in Figure 20.9.

Exercise 20.10 Answer the following questions briefly:

1. Explain how media recovery is handled in ARIES.
2. What are the pros and cons of using fuzzy dumps for media recovery?
3. What are the similarities and differences between checkpoints and fuzzy dumps?
4. Contrast ARIES with other WAL-based recovery schemes.
5. Contrast ARIES with shadow-page-based recovery.

BIBLIOGRAPHIC NOTES

Our discussion of the ARIES recovery algorithm is based on [473]. [241] is a survey article that contains a very readable, short description of ARIES. [470, 474] also discuss ARIES. Fine-granularity locking increases concurrency but at the cost of more locking activity; [471] suggests a technique based on LSNs for alleviating this problem. [396] presents a formal verification of ARIES.

[300] is an excellent survey that provides a broader treatment of recovery algorithms than our coverage, in which we have chosen to concentrate on one particular algorithm. [13] considers performance of concurrency control and recovery algorithms, taking into account their interactions. The impact of recovery on concurrency control is also discussed in [675]. [544] contains a performance analysis of various recovery techniques. [199] compares recovery techniques for main memory database systems, which are optimized for the case that most of the active data set fits in main memory. [288] provides an encyclopedic treatment of transaction processing.

[414] presents a description of a recovery algorithm based on write-ahead logging in which 'loser' transactions are first undone and then (only) transactions that committed before the crash are redone. Shadow paging is described in [428, 287]. A scheme that uses a combination of shadow paging and in-place updating is described in [543].

23

DECISION SUPPORT

> Nothing is more difficult, and therefore more precious, than to be able to decide.
>
> —Napoleon Bonaparte

Database management systems are widely used by organizations for maintaining data that documents their everyday operations. In applications that update such *operational data*, transactions typically make small changes (for example, adding a reservation or depositing a check) and a large number of transactions must be reliably and efficiently processed. Such **online transaction processing (OLTP)** applications have driven the growth of the DBMS industry in the past three decades and will doubtless continue to be important. DBMSs have traditionally been optimized extensively to perform well in such applications.

Recently, however, organizations have increasingly emphasized applications in which current and historical data are comprehensively analyzed and explored, identifying useful trends and creating summaries of the data, in order to support high-level decision making. Such applications are referred to as **decision support**. Decision support has rapidly grown into a multibillion dollar industry, and further growth is expected. A number of vendors offer specialized database systems and analysis tools to facilitate decision support. Industry organizations are emerging to set standards and create consensus on issues like language and architecture design.

Mainstream relational DBMS vendors have recognized the importance of this market segment and are adding features to their products in order to support it. In particular, novel indexing and query optimization techniques are being added to support complex queries. Systems are also providing additional features for defining and using views. The use of views has gained rapidly in popularity because of their utility in applications involving complex data analysis. While queries on views can be answered by evaluating the view definition when the query is submitted, views also offer the option of precomputing the view definition and thereby making queries run much faster. This option becomes increasingly attractive as the view definition increases in complexity and the frequency of queries increases.

Carrying the motivation for precomputed views one step further, organizations can consolidate information from several databases into a *data warehouse* by copying tables

from many sources into one location or by materializing a view that is defined over tables from several sources. Data warehousing has become widespread, and many specialized products are now available to create and manage warehouses of data from multiple databases.

We begin this chapter with an overview of decision support in Section 23.1. We cover data warehousing in Section 23.2 and present on-line analytic processing, or OLAP, in Section 23.3. We discuss implementation techniques to support OLAP in Section 23.4. These new implementation techniques form the basis for specialized OLAP products, and are also being added to relational DBMS products to support complex decision support applications. We discuss the role of views in decision support applications and techniques for rapidly processing queries on views in Section 23.5. Finally, in Section 23.6, we discuss a recent trend toward quickly computing approximate answers or a desired subset of answers, rather than computing all answers.

23.1 INTRODUCTION TO DECISION SUPPORT

Organizational decision making requires a comprehensive view of all aspects of an enterprise, and many organizations have therefore created consolidated **data warehouses** that contain data drawn from several databases maintained by different business units, together with historical and summary information.

The trend toward data warehousing is complemented by an increased emphasis on powerful analysis tools. There are many characteristics of decision support queries that make traditional SQL systems inadequate:

- The conditions in the `WHERE` clause often contain many `AND` and `OR` conditions. As we saw in Section 12.3.3, `OR` conditions, in particular, are poorly handled in many relational DBMSs.
- Applications require extensive use of statistical functions such as standard deviation, which are not supported in SQL-92. Thus, SQL queries must frequently be embedded in a host language program.
- Many queries involve conditions over time or require aggregating over time periods. SQL-92 provides poor support for such time-series analysis.
- Users often need to pose several related queries. Since there is no convenient way to express these commonly occurring families of queries, users have to write them as a collection of independent queries, which can be tedious. Further, the DBMS has no way to recognize and exploit optimization opportunities arising from executing many related queries together.

Three broad classes of analysis tools are available. First, there are systems that support a class of stylized queries that typically involve group-by and aggregation operators

and provide excellent support for complex boolean conditions, statistical functions, and features for time-series analysis. Applications dominated by such queries are called **online analytic processing**, or **OLAP**. These systems support a querying style in which the data is best thought of as a multidimensional array, and are influenced by end user tools such as spreadsheets, in addition to database query languages.

Second, there are DBMSs that support traditional SQL-style queries but are designed to also support OLAP queries efficiently. Such systems can be regarded as relational DBMSs optimized for decision support applications. Many vendors of relational DBMSs are currently enhancing their products in this direction, and over time the distinction between specialized OLAP systems and relational DBMSs enhanced to support OLAP queries is likely to diminish.

The third class of analysis tools is motivated by the desire to find interesting or unexpected trends and patterns in large data sets, rather than by the complex query characteristics listed above. In **exploratory data analysis**, although an analyst can recognize an 'interesting pattern' when shown such a pattern, it is very difficult to formulate a query that captures the essence of an interesting pattern. For example, an analyst looking at credit-card usage histories may want to detect unusual activity indicating misuse of a lost or stolen card. A catalog merchant may want to look at customer records to identify promising customers for a new promotion; this identification would depend on income levels, buying patterns, demonstrated interest areas, and so on. The amount of data in many applications is too large to permit manual analysis or even traditional statistical analysis, and the goal of **data mining** is to support exploratory analysis over very large data sets. We discuss data mining further in Chapter 24.

Clearly, evaluating OLAP or data mining queries over globally distributed data is likely to be excruciatingly slow. Further, for such complex analysis, often statistical in nature, it is not essential that the most current version of the data be used. The natural solution is to create a centralized repository of all the data, i.e., a data warehouse. Thus, the availability of a warehouse facilitates the application of OLAP and data mining tools, and conversely, the desire to apply such analysis tools is a strong motivation for building a data warehouse.

23.2 DATA WAREHOUSING

Data warehouses contain consolidated data from many sources, augmented with summary information and covering a long time period. Warehouses are much larger than other kinds of databases; sizes ranging from several gigabytes to terabytes are common. Typical workloads involve ad hoc, fairly complex queries and fast response times are important. These characteristics differentiate warehouse applications from OLTP applications, and different DBMS design and implementation techniques must be used

to achieve satisfactory results. A distributed DBMS with good scalability and high availability (achieved by storing tables redundantly at more than one site) is required for very large warehouses.

A typical data warehousing architecture is illustrated in Figure 23.1. An organiza-

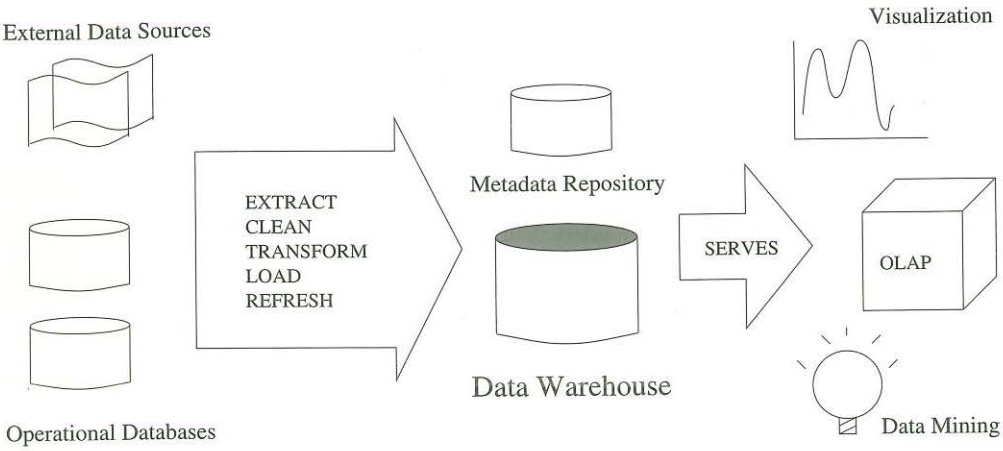


Figure 23.1 A Typical Data Warehousing Architecture

tion's daily operations access and modify **operational databases**. Data from these operational databases and other external sources (e.g., customer profiles supplied by external consultants) are **extracted** by using *gateways*, or standard external interfaces supported by the underlying DBMSs. A **gateway** is an application program interface that allows client programs to generate SQL statements to be executed at a server (see Section 5.10). Standards such as *Open Database Connectivity (ODBC)* and *Open Linking and Embedding for Databases (OLE-DB)* from Microsoft and *Java Database Connectivity (JDBC)* are emerging for gateways.

23.2.1 Creating and Maintaining a Warehouse

There are many challenges in creating and maintaining a large data warehouse. A good database schema must be designed to hold an integrated collection of data copied from diverse sources. For example, a company warehouse might include the inventory and personnel departments' databases, together with sales databases maintained by offices in different countries. Since the source databases are often created and maintained by different groups, there are a number of semantic mismatches across these databases, such as different currency units, different names for the same attribute, and differences in how tables are normalized or structured; these differences must be reconciled when data is brought into the warehouse. After the warehouse schema is designed, the

warehouse must be populated, and over time, it must be kept consistent with the source databases.

Data is **extracted** from operational databases and external sources, **cleaned** to minimize errors and fill in missing information when possible, and **transformed** to reconcile semantic mismatches. Transforming data is typically accomplished by defining a relational view over the tables in the data sources (the operational databases and other external sources). **Loading** data consists of materializing such views and storing them in the warehouse. Unlike a standard view in a relational DBMS, therefore, the view is stored in a database (the warehouse) that is different from the database(s) containing the tables it is defined over.

The cleaned and transformed data is finally **loaded** into the warehouse. Additional preprocessing such as sorting and generation of summary information is carried out at this stage. Data is partitioned and indexes are built for efficiency. Due to the large volume of data, loading is a slow process. Loading a terabyte of data sequentially can take weeks, and loading even a gigabyte can take hours. Parallelism is therefore important for loading warehouses.

After data is loaded into a warehouse, additional measures must be taken to ensure that the data in the warehouse is periodically **refreshed** to reflect updates to the data sources and to periodically **purge** data that is too old from the warehouse (perhaps onto archival media). Observe the connection between the problem of refreshing warehouse tables and asynchronously maintaining replicas of tables in a distributed DBMS. Maintaining replicas of source relations is an essential part of warehousing, and this application domain is an important factor in the popularity of asynchronous replication (Section 21.10.2), despite the fact that asynchronous replication violates the principle of distributed data independence. The problem of refreshing warehouse tables (which are materialized views over tables in the source databases) has also renewed interest in incremental maintenance of materialized views. (We discuss materialized views in Section 23.5.)

An important task in maintaining a warehouse is keeping track of the data currently stored in it; this bookkeeping is done by storing information about the warehouse data in the system catalogs. The system catalogs associated with a warehouse are very large and are often stored and managed in a separate database called a **metadata repository**. The size and complexity of the catalogs is in part due to the size and complexity of the warehouse itself and in part because a lot of administrative information must be maintained. For example, we must keep track of the source of each warehouse table and when it was last refreshed, in addition to describing its fields.

The value of a warehouse is ultimately in the analysis that it enables. The data in a warehouse is typically accessed and analyzed using a variety of tools, including OLAP

query engines, data mining algorithms, information visualization tools, statistical packages, and report generators.

23.3 OLAP

OLAP applications are dominated by ad hoc, complex queries. In SQL terms, these are queries that involve group-by and aggregation operators. The natural way to think about typical OLAP queries, however, is in terms of a multidimensional data model. We begin this section by presenting the multidimensional data model and comparing it with a relational representation of data. We describe OLAP queries in terms of the multidimensional data model and then consider some new implementation techniques designed to support such queries. Finally, we briefly contrast database design for OLAP applications with more traditional relational database design.

23.3.1 Multidimensional Data Model

In the multidimensional data model, the focus is on a collection of numeric **measures**. Each measure depends on a set of **dimensions**. We will use a running example based on sales data. The measure attribute in our example is *sales*. The dimensions are Product, Location, and Time. Given a product, a location, and a time, we have at most associated one sales value. If we identify a product by a unique identifier *pid*, and similarly identify location by *locid* and time by *timeid*, we can think of sales information as being arranged in a three-dimensional array Sales. This array is shown in Figure 23.2; for clarity, we show only the values for a single *locid* value, *locid*= 1, which can be thought of as a slice orthogonal to the *locid* axis.

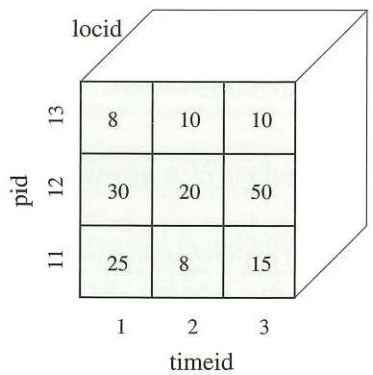


Figure 23.2 Sales: A Multidimensional Dataset

This view of data as a multidimensional array is readily generalized to more than three dimensions. In OLAP applications, the bulk of the data can be represented in such a multidimensional array. Indeed, some OLAP systems, for example, Essbase

from Arbor Software, actually store data in a multidimensional array (of course, implemented without the usual programming language assumption that the entire array fits in memory). OLAP systems that use arrays to store multidimensional datasets are called **multidimensional OLAP (MOLAP)** systems.

The data in a multidimensional array can also be represented as a relation, as illustrated in Figure 23.3, which shows the same data as in Figure 23.2; additional rows corresponding to the 'slice' *locid*= 2 are shown in addition to the data visible in Figure 23.3. This relation, which relates the dimensions to the measure of interest, is called the **fact table**.

locid	*city*	*state*	*country*
1	Ames	Iowa	USA
2	Chennai	TN	India
5	Tempe	Arizona	USA

Locations

pid	*pname*	*category*	*price*
11	Lee Jeans	Apparel	25
12	Zord	Toys	18
13	Biro Pen	Stationery	2

Products

pid	*timeid*	*locid*	*sales*
11	1	1	25
11	2	1	8
11	3	1	15
12	1	1	30
12	2	1	20
12	3	1	50
13	1	1	8
13	2	1	10
13	3	1	10
11	1	2	35
11	2	2	22
11	3	2	10
12	1	2	26
12	2	2	45
12	3	2	20
13	1	2	20
13	2	2	40
13	3	2	5

Sales

Figure 23.3 Locations, Products, and Sales Represented as Relations

Now let us turn to dimensions. Each dimension can have a set of associated attributes. For example, the Location dimension is identified by the *locid* attribute, which we used to identify a location in the Sales table. We will assume that it also has attributes

country, *state*, and *city*. We will also assume that the Product dimension has attributes *pname*, *category*, and *price*, in addition to the identifier *pid*. The *category* of a product indicates its general nature; for example, a product *pant* could have category value *apparel*. We will assume that the Time dimension has attributes *date*, *week*, *month*, *quarter*, *year*, and *holiday_flag*, in addition to the identifier *timeid*.

For each dimension, the set of associated values can be structured as a hierarchy. For example, cities belong to states, and states belong to countries. Dates belong to weeks and to months, both weeks and months are contained in quarters, and quarters are contained in years. (Note that a week could span a month; thus, weeks are not contained in months.) Some of the attributes of a dimension describe the position of a dimension value with respect to this underlying hierarchy of dimension values. The hierarchies for the Product, Location, and Time hierarchies in our example are shown at the attribute level in Figure 23.4.

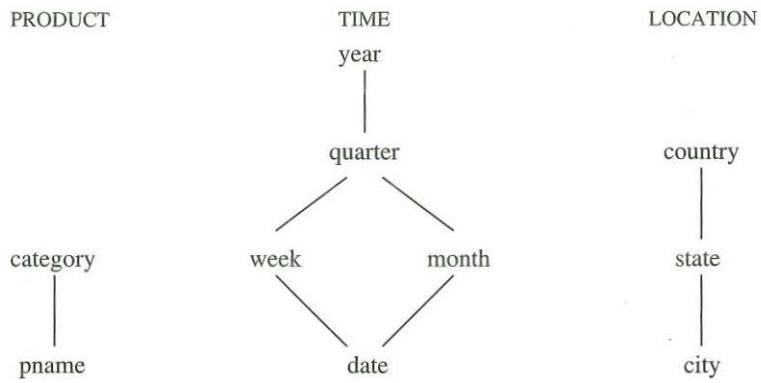


Figure 23.4 Dimension Hierarchies

Information about dimensions can also be represented as a collection of relations:

Locations(*locid:* `integer`, *city:* `string`, *state:* `string`, *country:* `string`)
Products(*pid:* `integer`, *pname:* `string`, *category:* `string`, *price:* `real`)
Times(*timeid:* `integer`, *date:* `string`, *week:* `integer`, *month:* `integer`,
quarter: `integer`, *year:* `integer`, *holiday_flag:* `boolean`)

These relations are much smaller than the fact table in a typical OLAP application; they are called the **dimension tables**. OLAP systems that store all information, including fact tables, as relations are called **relational OLAP (ROLAP)** systems.

The Times table illustrates the attention paid to the Time dimension in typical OLAP applications. SQL's date and timestamp data types are not adequate; in order to support summarizations that reflect business operations, information such as fiscal quarters, holiday status, and so on is maintained for each time value.

23.3.2 OLAP Queries

Now that we have seen the multidimensional model of data, let us consider how such data can be queried and manipulated. The operations supported by this model are strongly influenced by end user tools such as spreadsheets. The goal is to give end users who are not SQL experts an intuitive and powerful interface for common business-oriented analysis tasks. Users are expected to pose ad hoc queries directly, without relying on database application programmers. In this section we assume that the user is working with a multidimensional dataset and that each operation returns either a different presentation or summarization of this underlying dataset; the underlying dataset is always available for the user to manipulate, regardless of the level of detail at which it is currently viewed.

A very common operation is aggregating a measure over one or more dimensions. The following queries are typical:

- Find the total sales.
- Find total sales for each city.
- Find total sales for each state.
- Find the top five products ranked by total sales.

The first three queries can be expressed as SQL queries over the fact and dimension tables, but the last query cannot be expressed in SQL (although we can approximate it if we return answers in sorted order by total sales, using `ORDER BY`).

When we aggregate a measure on one or more dimensions, the aggregated measure depends on fewer dimensions than the original measure. For example, when we compute the total sales by city, the aggregated measure is *total sales* and it depends only on the Location dimension, whereas the original *sales* measure depended on the Location, Time, and Product dimensions.

Another use of aggregation is to summarize at different levels of a dimension hierarchy. If we are given total sales per city, we can aggregate on the Location dimension to obtain sales per state. This operation is called **roll-up** in the OLAP literature. The inverse of roll-up is **drill-down**: given total sales by state, we can ask for a more detailed presentation by drilling down on Location. We can ask for sales by city or just sales by city for a selected state (with sales presented on a per-state basis for the remaining states, as before). We can also drill down on a dimension other than Location. For example, we can ask for total sales for each product for each state, drilling down on the Product dimension.

Another common operation is **pivoting**. Consider a tabular presentation of the Sales table. If we pivot it on the Location and Time dimensions, we obtain a table of total

sales for each location for each time value. This information can be presented as a two-dimensional chart in which the axes are labeled with location and time values, and the entries in the chart correspond to the total sales for that location and time. Thus, values that appear in columns of the original presentation become labels of axes in the result presentation. Of course, pivoting can be combined with aggregation; we can pivot to obtain yearly sales by state. The result of pivoting is called a **cross-tabulation** and is illustrated in Figure 23.5. Observe that in spreadsheet style, in addition to the total sales by year and state (taken together), we also have additional summaries of sales by year and sales by state.

	WI	CA	Total
1995	63	81	144
1996	38	107	145
1997	75	35	110
Total	176	223	399

Figure 23.5 Cross-Tabulation of Sales by Year and State

Pivoting can also be used to change the dimensions of the cross-tabulation; from a presentation of sales by year and state, we can obtain a presentation of sales by product and year.

The Time dimension is very important in OLAP. Typical queries include:

- Find total sales by month.
- Find total sales by month for each city.
- Find the percentage change in the total monthly sales for each product.
- Find the trailing n day moving average of sales. (For each day, we must compute the average daily sales over the preceding n days.)

The first two queries can be expressed as SQL queries over the fact and dimension tables. The third query can be expressed too, but it is quite complicated in SQL. The last query cannot be expressed in SQL if n is to be a parameter of the query.

Clearly, the OLAP framework makes it convenient to pose a broad class of queries. It also gives catchy names to some familiar operations: **slicing** a dataset amounts to an equality selection on one or more dimensions, possibly also with some dimensions projected out. **Dicing** a dataset amounts to a range selection. These terms come from visualizing the effect of these operations on a cube or cross-tabulated representation of the data.

Comparison with SQL Queries

Some OLAP queries cannot be easily expressed, or cannot be expressed at all, in SQL, as we saw in the above discussion. Notably, queries that rank results and queries that involve time-oriented operations fall into this category.

A large number of OLAP queries, however, can be expressed in SQL. Typically, they involve grouping and aggregation, and a single OLAP operation leads to several closely related SQL queries. For example, consider the cross-tabulation shown in Figure 23.5, which was obtained by pivoting the Sales table. To obtain the same information, we would issue the following queries:

```
SELECT   SUM (S.sales)
FROM     Sales S, Times T, Locations L
WHERE    S.timeid=T.timeid AND S.locid=L.locid
GROUP BY T.year, L.state
```

This query generates the entries in the body of the chart (outlined by the dark lines). The summary row at the bottom is generated by the query:

```
SELECT   SUM (S.sales)
FROM     Sales S, Times T
WHERE    S.timeid=T.timeid
GROUP BY T.year
```

The summary column on the right is generated by the query:

```
SELECT   SUM (S.sales)
FROM     Sales S, Locations L
WHERE    S.locid=L.locid
GROUP BY L.state
```

The example cross-tabulation can be thought of as roll-up on the Location dimension, on the Time dimension, and on the Location and Time dimensions together. Each roll-up corresponds to a single SQL query with grouping. In general, given a measure with k associated dimensions, we can roll up on any subset of these k dimensions, and so we have a total of 2^k such SQL queries.

Through high-level operations such as pivoting, users can generate many of these 2^k SQL queries. Recognizing the commonalities between these queries enables more efficient, coordinated computation of the set of queries. A proposed extension to SQL called the `CUBE` is equivalent to a collection of `GROUP BY` statements, with one `GROUP BY` statement for each subset of the k dimensions. We illustrate it using the Sales relation. Consider the following query:

```
CUBE pid, locid, timeid BY SUM Sales
```

This query will roll up the table Sales on all eight subsets of the set {pid, locid, timeid} (including the empty subset). It is equivalent to eight queries of the form:

```
SELECT    SUM (S.sales)
FROM      Sales S
GROUP BY  grouping-list
```

The queries differ only in the *grouping-list*, which is some subset of the set {pid, locid, timeid}. We can think of these eight queries as being arranged in a lattice, as shown in Figure 23.6. The result tuples at a node can be aggregated further to compute the result for any child of the node. This relationship between the queries arising in a CUBE can be exploited for efficient evaluation.

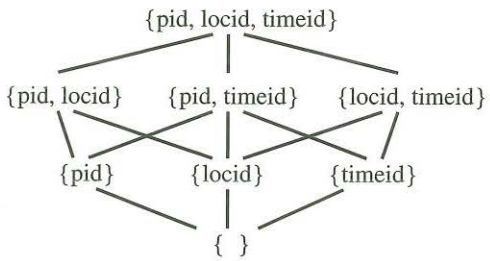


Figure 23.6 The Lattice of GROUP BY Queries in a CUBE Query

We conclude our discussion of the relationship between SQL and OLAP queries by noting that they complement each other, and both are important for decision support. The goal of OLAP is to enable end users to ask a broad class of business-oriented queries easily and with interactive response times over very large datasets. SQL, on the other hand, can be used to write complex queries that combine information from several relations. The data need not be schemas corresponding to the multidimensional data model, and the OLAP querying idioms are not always applicable. Such complex queries are written by application programmers, compiled, and made available to end users as 'canned' programs, often through a menu-driven graphical interface. The importance of such SQL applications is reflected in the increased attention being paid to optimizing complex SQL queries and the emergence of decision support oriented SQL benchmarks, such as TPC-D.

A Note on Statistical Databases

Many OLAP concepts are present in earlier work on **statistical databases (SDBs)**, which are database systems designed to support statistical applications, although this

connection has not been sufficiently recognized because of differences in application domains and terminology. The multidimensional data model, with the notions of a measure associated with dimensions, and classification hierarchies for dimension values, is also used in SDBs. OLAP operations such as roll-up and drill-down have counterparts in SDBs. Indeed, some implementation techniques developed for OLAP have also been applied to SDBs.

Nonetheless, there are some differences arising from the different domains that OLAP and SDBs were developed to support. For example, SDBs are used in socioeconomic applications, where classification hierarchies and privacy issues are very important. This is reflected in the fact that classification hierarchies in SDBs are more complex than in OLAP and have received more attention, along with issues such as potential breaches of privacy. (The privacy issue concerns whether a user with access to summarized data can reconstruct the original, unsummarized data.) In contrast, OLAP has been aimed at business applications with large volumes of data, and efficient handling of very large datasets has received more attention than in the SDB literature.

23.3.3 Database Design for OLAP

Figure 23.7 shows the tables in our running sales example.

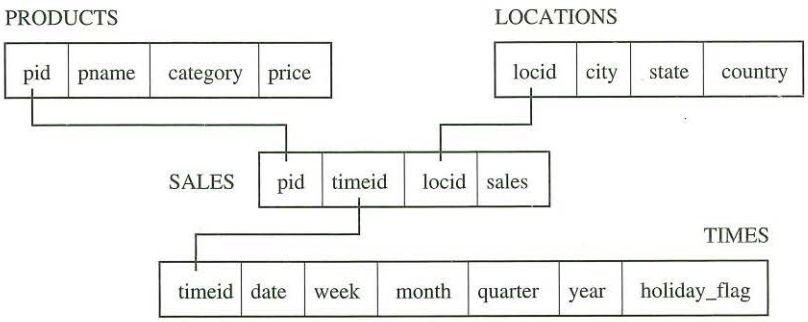


Figure 23.7 An Example of a Star Schema

It suggests a star, centered at the fact table Sales; such a combination of a fact table and dimension tables is called a **star schema**. This schema pattern is very common in databases designed for OLAP. The bulk of the data is typically in the fact table, which has no redundancy; it is usually in BCNF. In fact, to minimize the size of the fact table, dimension identifiers (such as *pid* and *timeid*) are system-generated identifiers.

Information about dimension values is maintained in the dimension tables. Dimension tables are usually not normalized. The rationale is that the dimension tables in a database used for OLAP are static and update, insertion, and deletion anomalies are not important. Further, because the size of the database is dominated by the fact table,

Beyond B+ trees: Complex queries have motivated the addition of powerful indexing techniques to DBMSs. In addition to B+ tree indexes, Oracle 8 supports bitmap and join indexes, and maintains these dynamically as the indexed relations are updated. Oracle 8 also supports indexes on expressions over attribute values, e.g., $10 * sal + bonus$. Microsoft SQL Server uses bitmap indexes. Sybase IQ supports several kinds of bitmap indexes, and may shortly add support for a linear hashing based index. Informix UDS supports R trees and Informix XPS supports bitmap indexes.

the space saved by normalizing dimension tables is negligible. Therefore, minimizing the computation time for combining facts in the fact table with dimension information is the main design criterion, which suggests that we avoid breaking a dimension table into smaller tables (which might lead to additional joins).

Small response times for interactive querying are important in OLAP, and most systems support the materialization of summary tables (typically generated through queries using grouping). Ad hoc queries posed by users are answered using the original tables along with precomputed summaries. A very important design issue is which summary tables should be materialized to achieve the best use of available memory and to answer commonly asked ad hoc queries with interactive response times. In current OLAP systems, deciding which summary tables to materialize may well be the most important design decision.

Finally, new storage structures and indexing techniques have been developed to support OLAP, and they present the database designer with additional physical design choices. We cover some of these implementation techniques briefly in the next section.

23.4 IMPLEMENTATION TECHNIQUES FOR OLAP

In this section we survey some implementation techniques motivated by the OLAP environment. The goal is to provide a feel for how OLAP systems differ from more traditional SQL systems; our discussion is far from comprehensive.

The mostly-read environment of OLAP systems makes the CPU overhead of maintaining indexes negligible, and the requirement of interactive response times for queries over very large datasets makes the availability of suitable indexes very important. This combination of factors has led to the development of new indexing techniques. We discuss several of these techniques. We then consider file organizations and other OLAP implementation issues briefly.

23.4.1 Bitmap Indexes

Consider a table that describes customers:

Customers(*custid:* `integer`, *name:* `string`, *gender:* `boolean`, *rating:* `integer`)

The *rating* value is an integer in the range 1 to 5, and only two values are recorded for *gender*. Columns with few possible values are called **sparse**. We can exploit sparsity to construct a new kind of index that greatly speeds up queries on these columns.

The idea is to record values for sparse columns as a sequence of bits, one for each possible value. For example, a *gender* value is either 10 or 01; a 1 in the first position denotes male, and 1 in the second position denotes female. Similarly, 10000 denotes the *rating* value 1, and 00001 denotes the *rating* value 5.

If we consider the *gender* values for all rows in the Customers table, we can treat this as a collection of two **bit vectors**, one of which has the associated value 'Male' and the other the associated value 'Female'. Each bit vector has one bit per row in the Customers table, indicating whether the value in that row is the value associated with the bit vector. The collection of bit vectors for a column is called a **bitmap index** for that column.

An example instance of the Customers table, together with the bitmap indexes for *gender* and *rating*, is shown in Figure 23.8.

M	*F*
1	0
1	0
0	1
1	0

custid	*name*	*gender*	*rating*
112	Joe	M	3
115	Ram	M	5
119	Sue	F	5
112	Woo	M	4

1	*2*	*3*	*4*	*5*
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

Figure 23.8 Bitmap Indexes on the Customers Relation

Bitmap indexes offer two important advantages over conventional hash and tree indexes. First, they allow the use of efficient bit operations to answer queries. For example, consider the query "How many male customers have a rating of 5?" We can take the first bit vector for *gender* and do a bit-wise `AND` with the fifth bit vector for *rating* to obtain a bit vector that has 1 for every male customer with rating 5. We can then count the number of 1s in this bit vector to answer the query. Second, bitmap indexes can be much more compact than a traditional B+ tree index and are very amenable to the use of compression techniques.

Bit vectors correspond closely to the rid-lists used to represent data entries in Alternative (3) for a traditional B+ tree index (see Section 8.3.1). In fact, we can think of a

bit vector for a given *age* value, say, as an alternative representation of the rid-list for that value. This leads to a possible way to combine bit vectors (and their advantages of bit-wise processing) with B+ tree indexes on columns that are not sparse; we can use Alternative (3) for data entries, using a bit vector representation of rid-lists. A caveat is that if an rid-list is very small, the bit vector representation may be much larger than a list of rid values, even if the bit vector is compressed. Further, the use of compression leads to decompression costs, offsetting some of the computational advantages of the bit vector representation.

23.4.2 Join Indexes

Computing joins with small response times is extremely hard for very large relations. One approach to this problem is to create an index that is designed to speed up specific join queries. Suppose that the Customers table is to be joined with a table called Purchases (recording purchases made by customers) on the *custid* field. We can create a collection of $\langle c, p \rangle$ pairs, where p is the rid of a Purchases record that joins with a Customers record with *custid* c.

This idea can be generalized to support joins over more than two relations. We will discuss the special case of a star schema, in which the fact table is likely to be joined with several dimension tables. Consider a join query that joins fact table F with dimension tables D1 and D2 and includes selection conditions on column C_1 of table D1 and column C_2 of table D2. We store a tuple $langler_1, r_2, r$ in the join index if r_1 is the rid of a tuple in table D1 with value c_1 in column C_1, r_2 is the rid of a tuple in table D2 with value c_2 in column C_2, and r is the rid of a tuple in the fact table F, and these three tuples join with each other.

The drawback of a join index is that the number of indexes can grow rapidly if several columns in each dimension table are involved in selections and joins with the fact table. An alternative kind of join index avoids this problem. Consider our example involving fact table F and dimension tables D1 and D2. Let C_1 be a column of D1 on which a selection is expressed in some query that joins D1 with F. Conceptually, we now join the F with D1 to extend the fields of F with the fields of D1, and index F on the 'virtual field' C_1: If a tuple of D1 with value c_1 in column C_1 joins with a tuple of F with rid r, we add a tuple $langlec_1, r$ to the join index. We create one such join index for each column of either D1 or D2 that involves a selection in some join with F; C_1 is an example of such a column.

The price paid with respect to the previous version of join indexes is that join indexes created in this way have to be combined (rid intersection) in order to deal with the join queries of interest to us. This can be done efficiently if we make the new indexes *bitmap* indexes; the result is called a **bitmapped join index**. The idea works especially well if columns such as C_1 are sparse, and therefore well suited to bitmap indexing.

Complex queries: The IBM DB2 optimizer recognizes star join queries and performs rid-based semijoins (using Bloom filters) to filter the fact table. Then fact table rows are rejoined to the dimension tables. Complex (multi-table) dimension queries (called 'snowflake queries') are supported. DB2 also supports CUBE using smart algorithms that minimize sorts. Microsoft SQL Server optimizes star join queries extensively. It considers taking the cross-product of small dimension tables before joining with the fact table, the use of join indexes, and rid-based semijoins. Oracle 8i also allows users to create dimensions to declare hierarchies and functional dependencies. It supports the CUBE operator and optimizes star join queries by eliminating joins when no column of a dimension table is part of the query result. There are also DBMS products developed specially for decision support applications, such as Sybase IQ and RedBrick (now part of Informix).

23.4.3 File Organizations

Since many OLAP queries involve just a few columns of a large relation, vertical partitioning becomes attractive. However, storing a relation column-wise can degrade performance for queries that involve several columns. An alternative in a mostly-read environment is to store the relation row-wise, but to also store each column separately.

A more radical file organization is to regard the fact table as a large multidimensional array, and to store it and index it as such. This approach is taken in MOLAP systems. Since the array is much larger than available main memory, it is broken up into contiguous chunks, as discussed in Section 25.7. In addition, traditional B+ tree indexes are created to enable quick retrieval of chunks that contain tuples with values in a given range for one or more dimensions.

23.4.4 Additional OLAP Implementation Issues

Our discussion of OLAP implementation techniques is far from complete. A number of other implementation issues must be considered for efficient OLAP query evaluation.

First, the use of compression is becoming widespread in database systems aimed at OLAP. The amount of information is so large that compression is obviously attractive. Further, the use of data structures like bitmap indexes, which are highly amenable to compression techniques, makes compression even more attractive.

Second, deciding which views to precompute and store in order to facilitate evaluation of ad hoc queries is a challenging problem. Especially for aggregate queries, the rich structure of operators such as `CUBE` offers many opportunities for a clever choice of views to precompute and store. Although the choice of views to precompute is made

by the database designer in current systems, ongoing research is aimed at automating this choice.

Third, many OLAP systems are enhancing query language and optimization features in novel ways. As an example of query language enhancement, Redbrick (recently acquired by Informix) supports a version of SQL that allows users to define new aggregation operators by writing code for *initialization*, *iteration*, and *termination*. For example, if tuples with fields *department*, *employee*, and *salary* are retrieved in sorted order by *department*, we can compute the standard deviation of salaries for each department; the initialization function would initialize the variables used to compute standard deviation, the iteration function would update the variables as each tuple is retrieved and processed, and the termination function would output the standard deviation for a department as soon as the first tuple for the next department is encountered. (Several ORDBMSs also support user-defined aggregate functions, and it is likely that this feature will be included in future versions of the SQL standard.) As an example of novel optimization features, some OLAP systems try to combine multiple scans, possibly part of different transactions, over a table. This seemingly simple optimization can be challenging: If the scans begin at different times, for example, we must keep track of the records seen by each scan to make sure that each scan sees every tuple exactly once; we must also deal with differences in the speeds at which the scan operations process tuples.

Finally, we note that the emphasis on query processing and decision support applications in OLAP systems is being complemented by a greater emphasis on evaluating complex SQL queries in traditional SQL systems. Traditional SQL systems are evolving to support OLAP-style queries more efficiently, incorporating techniques previously found only in specialized OLAP systems.

23.5 VIEWS AND DECISION SUPPORT

Views are widely used in decision support applications. Different groups of analysts within an organization are typically concerned with different aspects of the business, and it is convenient to define views that give each group insight into the business details that concern them. Once a view is defined, we can write queries or new view definitions that use it, as we saw in Section 3.6; in this respect a view is just like a base table. Evaluating queries posed against views is very important for decision support applications. In this section, we consider how such queries can be evaluated efficiently after placing views within the context of decision support applications.

23.5.1 Views, OLAP, and Warehousing

Views are closely related to OLAP and data warehousing.

Views and OLAP: OLAP queries are typically aggregate queries. Analysts want fast answers to these queries over very large datasets, and it is natural to consider precomputing views (see Sections 23.5.3 and 23.5.4). In particular, the CUBE operator—discussed in Section 23.3.2—gives rise to several aggregate queries that are closely related. The relationships that exist between the many aggregate queries that arise from a single CUBE operation can be exploited to develop very effective precomputation strategies. The idea is to choose a subset of the aggregate queries for materialization in such a way that typical CUBE queries can be quickly answered by using the materialized views and doing some additional computation. The choice of views to materialize is influenced by how many queries they can potentially speed up and by the amount of space required to store the materialized view (since we have to work with a given amount of storage space).

Views and Warehousing: A data warehouse is just a collection of asynchronously replicated tables and periodically maintained views. A warehouse is characterized by its size, the number of tables involved, and the fact that most of the underlying tables are from external, independently maintained databases. Nonetheless, the fundamental problem in warehouse maintenance is asynchronous maintenance of replicated tables and materialized views (see Section 23.5.4).

23.5.2 Query Modification

Consider the view RegionalSales, defined below, which computes sales of products by category and state:

```
CREATE VIEW RegionalSales (category, sales, state)
       AS SELECT P.category, S.sales, L.state
          FROM   Products P, Sales S, Locations L
          WHERE  P.pid = S.pid AND S.locid = L.locid
```

The following query computes the total sales for each category by state:

```
SELECT   R.category, R.state, SUM (R.sales)
FROM     RegionalSales R
GROUP BY R.category, R.state
```

While the SQL-92 standard does not specify how to evaluate queries on views, it is useful to think in terms of a process called **query modification**. The idea is to replace the occurrence of RegionalSales in the query by the view definition. The result on the above query is:

```
SELECT   R.category, R.state, SUM (R.sales)
FROM     ( SELECT P.category, S.sales, L.state
```

```
        FROM    Products P, Sales S, Locations L
        WHERE   P.pid = S.pid AND S.locid = L.locid ) AS R
GROUP BY R.category, R.state
```

23.5.3 View Materialization versus Computing on Demand

We can answer a query on a view by evaluating the modified query constructed using the query modification technique described above. Often, however, queries against complex view definitions must be answered very fast because users engaged in decision support activities require interactive response times. Even with sophisticated optimization and evaluation techniques, there is a limit to how fast we can answer such queries.

A popular approach to dealing with this problem is to evaluate the view definition and store the result. When a query is now posed on the view, the (unmodified) query is executed directly on the precomputed result. This approach is called **view materialization** and is likely to be much faster than the query modification approach because the complex view need not be evaluated when the query is computed. The drawback, of course, is that we must maintain the consistency of the precomputed (or *materialized*) view whenever the underlying tables are updated.

Consider the RegionalSales view. It involves a join of Sales, Products, and Locations and is likely to be expensive to compute. On the other hand, if it is materialized and stored with a clustered B+ tree index on the composite search key ⟨category, state, sales⟩, we can answer the example query by an index-only scan.

Given the materialized view and this index, we can also answer queries of the following form efficiently:

```
SELECT    R.state, SUM (R.sales)
FROM      RegionalSales R
WHERE     R.category = 'Laptop'
GROUP BY  R.state
```

To answer such a query, we can use the index on the materialized view to locate the first index leaf entry with *category* = 'Laptop' and then scan the leaf level until we come to the first entry with *category* not equal to 'Laptop.'

The given index is less effective on the following query, for which we are forced to scan the entire leaf level:

```
SELECT    R.state, SUM (R.sales)
FROM      RegionalSales R
```

```
WHERE     R.state = 'Wisconsin'
GROUP BY  R.category
```

This example indicates how the choice of views to materialize and the indexes to create are affected by the expected workload. This point is illustrated further by our next example.

Consider the following two queries:

```
SELECT    P.category, SUM (S.sales)
FROM      Products P, Sales S
WHERE     P.pid = S.pid
GROUP BY  P.category

SELECT    L.state, SUM (S.sales)
FROM      Locations L, Sales S
WHERE     L.locid = S.locid
GROUP BY  L.state
```

The above two queries require us to join the Sales table (which is likely to be very large) with another table and to then aggregate the result. How can we use materialization to speed these queries up? The straightforward approach is to precompute each of the joins involved (Products with Sales and Locations with Sales) or to precompute each query in its entirety. An alternative approach is to define the following view:

```
CREATE    VIEW TotalSales (pid, locid, total)
          AS SELECT    S.pid, S.locid, SUM (S.sales)
             FROM      Sales S
             GROUP BY  S.pid, S.locid
```

The view TotalSales can be materialized and used instead of Sales in our two example queries:

```
SELECT    P.category, SUM (T.total)
FROM      Products P, TotalSales T
WHERE     P.pid = T.pid
GROUP BY  P.category

SELECT    L.state, SUM (T.total)
FROM      Locations L, TotalSales T
WHERE     L.locid = T.locid
GROUP BY  L.state
```

23.5.4 Issues in View Materialization

There are three main questions to consider with regard to view materialization:

1. What views should we materialize and what indexes should we build on the materialized views?
2. Given a query on a view and a set of materialized views, can we exploit the materialized views to answer the query?
3. How frequently should we refresh materialized views in order to make them consistent with changes to the underlying tables?

As the example queries using TotalSales illustrated, the answers to the first two questions are related. The choice of views to materialize and index is governed by the expected workload, and the discussion of indexing in Chapter 16 is relevant to this question as well. The choice of views to materialize is more complex than just choosing indexes on a set of database tables, however, because the range of alternative views to materialize is wider. The goal is to materialize a small, carefully chosen set of views that can be utilized to quickly answer most of the important queries. Conversely, once we have chosen a set of views to materialize, we have to consider how they can be used to answer a given query.

A materialized view is said to be **refreshed** when we make it consistent with changes to its underlying tables. Ideally, algorithms for refreshing a view should be **incremental** in that the cost is proportional to the extent of the change, rather than the cost of recomputing the view from scratch. While it is usually possible to incrementally refresh views when new tuples are added to the underlying tables, incremental refreshing is harder when tuples are deleted from the underlying tables.

A **view maintenance policy** is a decision about when a view is refreshed and is independent of whether the refresh is incremental or not. A view can be refreshed within the same transaction that updates the underlying tables. This is called **immediate view maintenance**. The update transaction is slowed by the refresh step, and the impact of refresh increases with the number of materialized views that depend upon the updated table.

Alternatively, we can defer refreshing the view. Updates are captured in a log and applied subsequently to the materialized views. There are several **deferred view maintenance policies**:

1. **Lazy:** The materialized view V is refreshed at the time a query is evaluated using V, if V is not already consistent with its underlying base tables. This approach slows down queries rather than updates, in contrast to immediate view maintenance.

Views for decision support: DBMS vendors are enhancing their main relational products to support decision support queries. IBM DB2 supports materialized views with transaction-consistent or user-invoked maintenance. Microsoft SQL Server supports **partition views**, which are unions of (many) horizontal partitions of a table. These are aimed at a warehousing environment where each partition could be, for example, a monthly update. Queries on partition views are optimized so that only relevant partitions are accessed. Oracle 8i supports materialized views with transaction-consistent, user-invoked, or time-scheduled maintenance.

2. **Periodic:** The materialized view is refreshed periodically, e.g., once a day. The discussion of the Capture and Apply steps in asynchronous replication (see Section 21.10.2) should be reviewed at this point since it is very relevant to periodic view maintenance. In fact, many vendors are extending their asynchronous replication features to support materialized views. Materialized views that are refreshed periodically are also called **snapshots**.

3. **Forced:** The materialized view is refreshed after a certain number of changes have been made to the underlying tables.

In periodic and forced view maintenance, queries may see an instance of the materialized view that is not consistent with the current state of the underlying tables. That is, the queries would see a different set of tuples if the view definition was recomputed. This is the price paid for fast updates and queries, and the trade-off is similar to the trade-off made in using asynchronous replication.

23.6 FINDING ANSWERS QUICKLY

A recent trend, fueled in part by the popularity of the Internet, is an emphasis on queries for which a user wants only the first few, or the 'best' few, answers quickly. When users pose queries to a search engine such as AltaVista, they rarely look beyond the first or second page of results. If they do not find what they are looking for, they refine their query and resubmit it. The same phenomenon is being observed in decision support applications, and some DBMS products (e.g., DB2) already support extended SQL constructs to specify such queries. A related trend is that for complex queries, users would like to see an approximate answer quickly and then have it be continually refined, rather than wait until the exact answer is available. We now discuss these two trends briefly.

23.6.1 Top N Queries

An analyst often wants to identify the top-selling handful of products, for example. We can sort by sales for each product and return answers in this order. If we have a million products and the analyst is interested in the top 10, this straightforward evaluation strategy is clearly wasteful, and it desirable for users to be able to explicitly indicate how many answers they want. This would enable the DBMS to optimize execution. The example query below asks for the top 10 products ordered by sales in a given location and time:

```
SELECT    P.pid, P.pname, S.sales
FROM      Sales S, Products P
WHERE     S.pid=P.pid AND S.locid=1 AND S.timeid=3
ORDER BY  S.sales DESC
OPTIMIZE  FOR 10 ROWS
```

The `OPTIMIZE FOR` N `ROWS` construct is not in SQL-92 (or even SQL:1999), but it is supported in IBM's DB2 product, and other products (e.g., Oracle 7) have similar constructs. In the absence of a cue such as `OPTIMIZE FOR 10 ROWS`, the DBMS computes sales for all products and returns them in descending order by sales. The application can close the result cursor (i.e., terminate the query execution) after consuming 10 rows, but considerable effort has already been expended in computing sales for all products and sorting them.

Now let us consider how a DBMS can make use of the `OPTIMIZE FOR` cue to execute the query efficiently. The key is to somehow compute sales only for products that are likely to be in the top 10 by sales. Suppose that we know the distribution of sales values because we maintain a histogram on the *sales* column of the Sales relation. We can then choose a value of *sales*, say c, such that only 10 products have a larger sales value. For those Sales tuples that meet this condition, we can apply the location and time conditions as well and sort the result. Evaluating the following query is equivalent to this approach:

```
SELECT    P.pid, P.pname, S.sales
FROM      Sales S, Products P
WHERE     S.pid=P.pid AND S.locid=1 AND S.timeid=3 AND S.sales > c
ORDER BY  S.sales DESC
```

This approach is, of course, much faster than the alternative of computing all product sales and sorting them, but there are some important problems to resolve:

1. *How do we choose the sales cutoff value c?* Histograms and other system statistics can be used for this purpose, but this can be a tricky issue. For one thing, the statistics maintained by a DBMS are only approximate. For another, even if we

choose the cutoff to reflect the top 10 sales values accurately, other conditions in the query may eliminate some of the selected tuples, leaving us with fewer than 10 tuples in the result.

2. *What if we have more than 10 tuples in the result?* Since the choice of the cutoff c is approximate, we could get more than the desired number of tuples in the result. This is easily handled by returning just the top 10 to the user. We have still saved considerably with respect to the approach of computing sales for all products, thanks to the conservative pruning of irrelevant sales information, using the cutoff c.

3. *What if we have fewer than 10 tuples in the result?* Even if we choose the sales cutoff c conservatively, there is the possibility that we compute fewer than 10 result tuples. In this case, we can re-execute the query with a smaller cutoff value c_2, or simply re-execute the original query with no cutoff.

The effectiveness of the approach depends on how well we can estimate the cutoff, and in particular, on minimizing the number of times we obtain fewer than the desired number of result tuples.

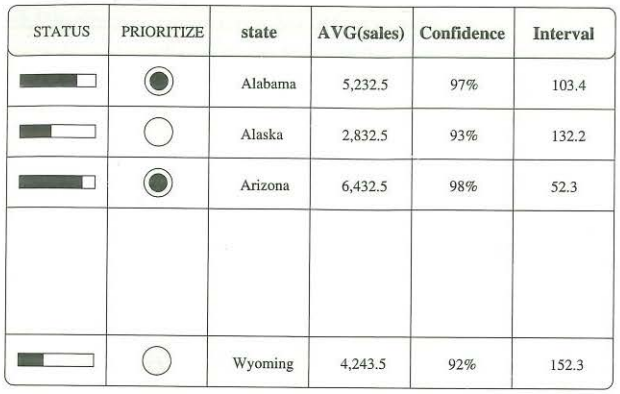
23.6.2 Online Aggregation

Consider the following query, which asks for the average sales amount by state:

```
SELECT    L.state, AVG (S.sales)
FROM      Sales S, Locations L
WHERE     S.locid=L.locid
GROUP BY L.state
```

This can be an expensive query if Sales and Locations are large relations, and we cannot achieve fast response times with the traditional approach of computing the anwer in its entirety when the query is presented. One alternative, as we have seen, is to use precomputation. Another alternative is to compute the answer to the query when the query is presented, but to return an approximate answer to the user as soon as possible. As the computation progresses, the answer quality is continually refined. This approach is called **online aggregation**. It is very attractive for queries involving aggregation, because efficient techniques for computing and refining approximate answers are available.

Online aggregation is illustrated in Figure 23.9: For each state—the grouping criterion for our example query—the current value for average sales is displayed, together with a confidence interval. The entry for Alaska tells us that the current estimate of average per-store sales in Alaska is $2,832.50, and that this is within the range $2,700.30 and $2,964.70 with 93 percent probability. The status bar in the first column indicates how



STATUS	PRIORITIZE	state	AVG(sales)	Confidence	Interval
		Alabama	5,232.5	97%	103.4
		Alaska	2,832.5	93%	132.2
		Arizona	6,432.5	98%	52.3
		Wyoming	4,243.5	92%	152.3

Figure 23.9 Online Aggregation

close we are to arriving at an exact value for the average sales, and the second column indicates whether calculating the average sales for this state is a priority. Estimating average sales for Alaska is not a priority, but estimating it for Arizona is a priority. As the figure indicates, the DBMS devotes more system resources to estimating the average sales for prioritized states; the estimate for Arizona is much tighter than that for Alaska, and holds with a higher probability. Users can set the priority for a state by clicking on the priority button at any time during the execution. This degree of interactivity, together with the continuous feedback provided by the visual display, makes online aggregation an attractive technique.

In order to implement online aggregation, a DBMS must incorporate statistical techniques to provide confidence intervals for approximate answers and use **nonblocking algorithms** for the relational operators. An algorithm is said to block if it does not produce output tuples until it has consumed all of its input tuples. For example, the sort-merge join algorithm blocks because sorting requires all input tuples before determining the first output tuple. Nested loops join and hash join are therefore preferable to sort-merge join for online aggregation. Similarly, hash-based aggregation is better than sort-based aggregation.

23.7 POINTS TO REVIEW

- A *data warehouse* contains consolidated data drawn from several different databases together with historical and summary information. *Online analytic processing (OLAP)* applications and *data mining* applications generate complex queries that make traditional SQL systems inadequate. Such applications support high-level decision making and are also called *decision support applications.* **(Section 23.1)**

- Information about daily operations of an organization is stored in *operational databases*. This data is *extracted* through *gateways*, then *cleaned* and *transformed* before *loading* it into the data warehouse. Data in the data warehouse is periodically *refreshed* to reflect updates, and it is periodically *purged* to delete outdated information. The system catalogs of the data warehouse can be very large and are managed in a separate database called the *metadata repository*. **(Section 23.2)**

- The multidimensional data model consists of *measures* and *dimensions*. The relation that relates the dimensions to the measures is called the *fact table*. OLAP systems that store multidimensional datasets as arrays are called *multidimensional OLAP (MOLAP)* systems. OLAP systems that store the data in relations are called *relational OLAP (ROLAP)* systems. Common OLAP operations have received special names: roll-up, drill-down, pivoting, slicing, and dicing. Databases designed for OLAP queries commonly arrange the fact and dimension tables in a *star schema*. **(Section 23.3)**

- Index structures that are especially suitable for OLAP systems include *bitmap indexes* and *join indexes*. **(Section 23.4)**

- Views are widely used in decision support applications. Since decision support systems require fast response times for interactive queries, queries involving views must be evaluated very efficiently. Views can either be materialized or computed on demand. We say that a materialized view is *refreshed* when we make it consistent with changes to the underlying tables. An algorithm for refreshing a view is *incremental* if the update cost is proportional to the amount of change at the base tables. A *view maintenance policy* determines when a view is refreshed. In *immediate view maintenance* the view is updated within the same transaction that modifies the underlying tables; otherwise the policy is said to be *deferred view maintenance*. Deferred view maintenance has three variants: In *lazy* maintenance we refresh the view at query time. In *periodic* maintenance we refresh the view periodically; such views are also called *snapshots*. In *forced* maintenance we refresh the view after a certain number of changes have been made to the base tables. **(Section 23.5)**

- New query paradigms include *top N queries* and *online aggregation*. In top N queries we only want to retrieve the first N rows of the query result. An online aggregation query returns an approximate answer to an aggregation query immediately and refines the answer progressively. **(Section 23.6)**

EXERCISES

Exercise 23.1 Briefly answer the following questions.

1. How do warehousing, OLAP, and data mining complement each other?

2. What is the relationship between data warehousing and data replication? Which form of replication (synchronous or asynchronous) is better suited for data warehousing? Why?
3. What is the role of the metadata repository in a data warehouse? How does it differ from a catalog in a relational DBMS?
4. What are the considerations in designing a data warehouse?
5. Once a warehouse is designed and loaded, how is it kept current with respect to changes to the source databases?
6. One of the advantages of a warehouse is that we can use it to track how the contents of a relation change over time; in contrast, we have only the current snapshot of a relation in a regular DBMS. Discuss how you would maintain the history of a relation R, taking into account that 'old' information must somehow be purged to make space for new information.
7. Describe dimensions and measures in the multidimensional data model.
8. What is a fact table, and why is it so important from a performance standpoint?
9. What is the fundamental difference between MOLAP and ROLAP systems?
10. What is a star schema? Is it typically in BCNF? Why or why not?
11. How is data mining different from OLAP?

Exercise 23.2 Consider the instance of the Sales relation shown in Figure 23.3.

1. Show the result of pivoting the relation on *pid* and *timeid*.
2. Write a collection of SQL queries to obtain the same result as in the previous part.
3. Show the result of pivoting the relation on *pid* and *locid*.

Exercise 23.3 Consider the cross-tabulation of the Sales relation shown in Figure 23.5.

1. Show the result of roll-up on *locid* (i.e., state).
2. Write a collection of SQL queries to obtain the same result as in the previous part.
3. Show the result of roll-up on *locid* followed by drill-down on *pid*.
4. Write a collection of SQL queries to obtain the same result as in the previous part, starting with the cross-tabulation shown in Figure 23.5.

Exercise 23.4 Consider the Customers relation and the bitmap indexes shown in Figure 23.8.

1. For the same data, if the underlying set of rating values is assumed to range from 1 to 10, show how the bitmap indexes would change.
2. How would you use the bitmap indexes to answer the following queries? If the bitmap indexes are not useful, explain why.
   (a) How many customers with a rating less than 3 are male?
   (b) What percentage of customers are male?
   (c) How many customers are there?

(d) How many customers are named Woo?

(e) Find the rating value with the greatest number of customers and also find the number of customers with that rating value; if several rating values have the maximum number of customers, list the requested information for all of them. (Assume that very few rating values have the same number of customers.)

Exercise 23.5 In addition to the Customers table of Figure 23.8 with bitmap indexes on *gender* and *rating*, assume that you have a table called Prospects, with fields *rating* and *prospectid*. This table is used to identify potential customers.

1. Suppose that you also have a bitmap index on the *rating* field of Prospects. Discuss whether or not the bitmap indexes would help in computing the join of Customers and Prospects on *rating*.
2. Suppose that you do *not* have a bitmap index on the *rating* field of Prospects. Discuss whether or not the bitmap indexes on Customers would help in computing the join of Customers and Prospects on *rating*.
3. Describe the use of a join index to support the join of these two relations with the join condition *custid=prospectid*.

Exercise 23.6 Consider the instances of the Locations, Products, and Sales relations shown in Figure 23.3.

1. Consider the basic join indexes described in Section 23.4.2. Suppose you want to optimize for the following two kinds of queries: Query 1 finds sales in a given city, and Query 2 finds sales in a given state. Show the indexes that you would create on the example instances shown in Figure 23.3.
2. Consider the bitmapped join indexes described in Section 23.4.2. Suppose you want to optimize for the following two kinds of queries: Query 1 finds sales in a given city, and Query 2 finds sales in a given state. Show the indexes that you would create on the example instances shown in Figure 23.3.
3. Consider the basic join indexes described in Section 23.4.2. Suppose you want to optimize for the following two kinds of queries: Query 1 finds sales in a given city for a given product name, and Query 2 finds sales in a given state for a given product category. Show the indexes that you would create on the example instances shown in Figure 23.3.
4. Consider the bitmapped join indexes described in Section 23.4.2. Suppose you want to optimize for the following two kinds of queries: Query 1 finds sales in a given city for a given product name, and Query 2 finds sales in a given state for a given product category. Show the indexes that you would create on the example instances shown in Figure 23.3.

Exercise 23.7 Consider the view NumReservations defined below:

```
CREATE VIEW NumReservations (sid, sname, numres)
      AS SELECT S.sid, S.sname, COUNT (*)
         FROM     Sailors S, Reserves R
         WHERE    S.sid = R.sid
         GROUP BY S.sid, S.sname
```

1. How is the following query, which is intended to find the highest number of reservations made by some one sailor, rewritten using query modification?

```
SELECT   MAX (N.numres)
FROM     NumReservations N
```

2. Consider the alternatives of computing on demand and view materialization for the above query. Discuss the pros and cons of materialization.
3. Discuss the pros and cons of materialization for the following query:

```
SELECT   N.sname, MAX (N.numres)
FROM     NumReservations N
GROUP BY N.sname
```

BIBLIOGRAPHIC NOTES

A good survey of data warehousing and OLAP is presented in [137], which is the source of Figure 23.1. [597] provides an overview of OLAP and statistical database research, showing the strong parallels between concepts and research in these two areas. The book by Kimball [374], one of the pioneers in warehousing, and the collection of papers [51] offer a good practical introduction to the area. The term OLAP was popularized by Codd's paper [160]. For a recent discussion of the performance of algorithms utilizing bitmap and other nontraditional index structures, see [500].

[624] discusses how queries on views can be converted to queries on the underlying tables through query modification. [308] compares the performance of query modification versus immediate and deferred view maintenance. [618] presents an analytical model of materialized view maintenance algorithms. A number of papers discuss how materialized views can be incrementally maintained as the underlying relations are changed. This area has become very active recently, in part because of the interest in *data warehouses*, which can be thought of as collections of views over relations from various sources. An excellent overview of the state of the art can be found in [293], which contains a number of influential papers together with additional material that provides context and background. The following partial list should provide pointers for further reading: [87, 161, 162, 294, 312, 498, 524, 553, 577, 616, 700].

[285] introduced the CUBE operator, and optimization of CUBE queries and efficient maintenance of the result of a CUBE query have been addressed in several papers, including [9, 81, 182, 310, 320, 389, 552, 556, 598, 699]. Related algorithms for processing queries with aggregates and grouping are presented in [136, 139]. [538] addresses the implementation of queries involving generalized quantifiers such as *a majority of*. [619] describes an access method to support processing of aggregate queries.

[114, 115] discuss how to evaluate queries for which only the first few answers are desired.L [192] considers how a probabilistic approach to query optimization can be applied to this problem. [316, 40] discuss how to return approximate answers to aggregate queries and to refine them 'online.'

25

OBJECT-DATABASE SYSTEMS

with Joseph M. Hellerstein
U. C. Berkeley

You know my methods, Watson. Apply them.

—Arthur Conan Doyle, *The Memoirs of Sherlock Holmes*

Relational database systems support a small, fixed collection of data types (e.g., integers, dates, strings), which has proven adequate for traditional application domains such as administrative data processing. In many application domains, however, much more complex kinds of data must be handled. Typically this complex data has been stored in OS file systems or specialized data structures, rather than in a DBMS. Examples of domains with complex data include computer-aided design and modeling (CAD/CAM), multimedia repositories, and document management.

As the amount of data grows, the many features offered by a DBMS—for example, reduced application development time, concurrency control and recovery, indexing support, and query capabilities—become increasingly attractive and, ultimately, necessary. In order to support such applications, a DBMS must support complex data types. Object-oriented concepts have strongly influenced efforts to enhance database support for complex data and have led to the development of object-database systems, which we discuss in this chapter.

Object-database systems have developed along two distinct paths:

- **Object-oriented database systems:** Object-oriented database systems are proposed as an alternative to relational systems and are aimed at application domains where complex objects play a central role. The approach is heavily influenced by object-oriented programming languages and can be understood as an attempt to add DBMS functionality to a programming language environment.
- **Object-relational database systems:** Object-relational database systems can be thought of as an attempt to extend relational database systems with the functionality necessary to support a broader class of applications and, in many ways, provide a bridge between the relational and object-oriented paradigms.

We will use acronyms for relational database management systems (**RDBMS**), object-oriented database management systems (**OODBMS**), and object-relational database management systems (**ORDBMS**). In this chapter we focus on ORDBMSs and emphasize how they can be viewed as a development of RDBMSs, rather than as an entirely different paradigm.

The SQL:1999 standard is based on the ORDBMS model, rather than the OODBMS model. The standard includes support for many of the complex data type features discussed in this chapter. We have concentrated on developing the fundamental concepts, rather than on presenting SQL:1999; some of the features that we discuss are not included in SQL:1999. We have tried to be consistent with SQL:1999 for notation, although we have occasionally diverged slightly for clarity. It is important to recognize that the main concepts discussed are common to both ORDBMSs and OODBMSs, and we discuss how they are supported in the ODL/OQL standard proposed for OODBMSs in Section 25.8.

RDBMS vendors, including IBM, Informix, and Oracle, are adding ORDBMS functionality (to varying degrees) in their products, and it is important to recognize how the existing body of knowledge about the design and implementation of relational databases can be leveraged to deal with the ORDBMS extensions. It is also important to understand the challenges and opportunities that these extensions present to database users, designers, and implementors.

In this chapter, sections 25.1 through 25.5 motivate and introduce object-oriented concepts. The concepts discussed in these sections are common to both OODBMSs and ORDBMSs, even though our syntax is similar to SQL:1999. We begin by presenting an example in Section 25.1 that illustrates why extensions to the relational model are needed to cope with some new application domains. This is used as a running example throughout the chapter. We discuss how abstract data types can be defined and manipulated in Section 25.2 and how types can be composed into structured types in Section 25.3. We then consider objects and object identity in Section 25.4 and inheritance and type hierarchies in Section 25.5.

We consider how to take advantage of the new object-oriented concepts to do ORDBMS database design in Section 25.6. In Section 25.7, we discuss some of the new implementation challenges posed by object-relational systems. We discuss ODL and OQL, the standards for OODBMSs, in Section 25.8, and then present a brief comparison of ORDBMSs and OODBMSs in Section 25.9.

25.1 MOTIVATING EXAMPLE

As a specific example of the need for object-relational systems, we focus on a new business data processing problem that is both harder and (in our view) more entertaining

than the dollars and cents bookkeeping of previous decades. Today, companies in industries such as entertainment are in the business of selling *bits*; their basic corporate assets are not tangible products, but rather software artifacts such as video and audio.

We consider the fictional Dinky Entertainment Company, a large Hollywood conglomerate whose main assets are a collection of cartoon characters, especially the cuddly and internationally beloved Herbert the Worm. Dinky has a number of Herbert the Worm films, many of which are being shown in theaters around the world at any given time. Dinky also makes a good deal of money licensing Herbert's image, voice, and video footage for various purposes: action figures, video games, product endorsements, and so on. Dinky's database is used to manage the sales and leasing records for the various Herbert-related products, as well as the video and audio data that make up Herbert's many films.

25.1.1 New Data Types

A basic problem confronting Dinky's database designers is that they need support for considerably richer data types than is available in a relational DBMS:

- **User-defined abstract data types (ADTs):** Dinky's assets include Herbert's image, voice, and video footage, and these must be stored in the database. Further, we need special functions to manipulate these objects. For example, we may want to write functions that produce a compressed version of an image or a lower-resolution image. (See Section 25.2.)
- **Structured types:** In this application, as indeed in many traditional business data processing applications, we need new types built up from atomic types using constructors for creating sets, tuples, arrays, sequences, and so on. (See Section 25.3.)
- **Inheritance:** As the number of data types grows, it is important to recognize the commonality between different types and to take advantage of it. For example, compressed images and lower-resolution images are both, at some level, just images. It is therefore desirable to *inherit* some features of image objects while defining (and later manipulating) compressed image objects and lower-resolution image objects. (See Section 25.5.)

How might we address these issues in an RDBMS? We could store images, videos, and so on as *BLOBs* in current relational systems. A **binary large object (BLOB)** is just a long stream of bytes, and the DBMS's support consists of storing and retrieving BLOBs in such a manner that a user does not have to worry about the size of the BLOB; a BLOB can span several pages, unlike a traditional attribute. All further processing of the BLOB has to be done by the user's application program, in the host language in which the SQL code is embedded. This solution is not efficient because we

Large objects in SQL: SQL:1999 includes a new data type called `LARGE OBJECT` or `LOB`, with two variants called `BLOB` (binary large object) and `CLOB` (character large object). This standardizes the large object support found in many current relational DBMSs. LOBs cannot be included in primary keys, `GROUP BY`, or `ORDER BY` clauses. They can be compared using equality, inequality, and substring operations. A LOB has a **locator** that is essentially a unique id and allows LOBs to be manipulated without extensive copying.
LOBs are typically stored separately from the data records in whose fields they appear. IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support LOBs.

are forced to retrieve all BLOBs in a collection even if most of them could be filtered out of the answer by applying user-defined functions (within the DBMS). It is not satisfactory from a data consistency standpoint either, because the semantics of the data is now heavily dependent on the host language application code and cannot be enforced by the DBMS.

As for structured types and inheritance, there is simply no support in the relational model. We are forced to map data with such complex structure into a collection of flat tables. (We saw examples of such mappings when we discussed the translation from ER diagrams with inheritance to relations in Chapter 2.)

This application clearly requires features that are not available in the relational model. As an illustration of these features, Figure 25.1 presents SQL:1999 DDL statements for a portion of Dinky's ORDBMS schema that will be used in subsequent examples. Although the DDL is very similar to that of a traditional relational system, it has some important distinctions that highlight the new data modeling capabilities of an ORDBMS. A quick glance at the DDL statements is sufficient for now; we will study them in detail in the next section, after presenting some of the basic concepts that our sample application suggests are needed in a next-generation DBMS.

25.1.2 Manipulating the New Kinds of Data

Thus far, we have described the new kinds of data that must be stored in the Dinky database. We have not yet said anything about how to *use* these new types in queries, so let's study two queries that Dinky's database needs to support. The syntax of the queries is not critical; it is sufficient to understand what they express. We will return to the specifics of the queries' syntax as we proceed.

Our first challenge comes from the Clog breakfast cereal company. Clog produces a cereal called Delirios, and it wants to lease an image of Herbert the Worm in front of

```
1. CREATE TABLE Frames
     (frameno integer, image jpeg_image, category integer);
2. CREATE TABLE Categories
     (cid integer, name text, lease_price float, comments text);
3. CREATE TYPE theater_t AS
     ROW(tno integer, name text, address text, phone text);
4. CREATE TABLE Theaters OF theater_t;
5. CREATE TABLE Nowshowing
     (film integer, theater ref(theater_t) with scope Theaters, start date, end date);
6. CREATE TABLE Films
     (filmno integer, title text, stars setof(text),
     director text, budget float);
7. CREATE TABLE Countries
     (name text, boundary polygon, population integer, language text);
```

Figure 25.1 SQL:1999 DDL Statements for Dinky Schema

a sunrise, to incorporate in the Delirios box design. A query to present a collection of possible images and their lease prices can be expressed in SQL-like syntax as in Figure 25.2. Dinky has a number of methods written in an imperative language like Java and registered with the database system. These methods can be used in queries in the same way as built-in methods, such as $=, +, -, <, >$, are used in a relational language like SQL. The *thumbnail* method in the `Select` clause produces a small version of its full-size input image. The *is_sunrise* method is a boolean function that analyzes an image and returns *true* if the image contains a sunrise; the *is_herbert* method returns *true* if the image contains a picture of Herbert. The query produces the frame code number, image thumbnail, and price for all frames that contain Herbert and a sunrise.

```
SELECT F.frameno, thumbnail(F.image), C.lease_price
FROM   Frames F, Categories C
WHERE  F.category = C.cid AND is_sunrise(F.image) AND is_herbert(F.image)
```

Figure 25.2 Extended SQL to Find Pictures of Herbert at Sunrise

The second challenge comes from Dinky's executives. They know that Delirios is exceedingly popular in the tiny country of Andorra, so they want to make sure that a number of Herbert films are playing at theaters near Andorra when the cereal hits the shelves. To check on the current state of affairs, the executives want to find the names of all theaters showing Herbert films within 100 kilometers of Andorra. Figure 25.3 shows this query in an SQL-like syntax.

```
SELECT N.theater->name, N.theater->address, F.title
FROM   Nowshowing N, Films F, Countries C
WHERE  N.film = F.filmno AND
       overlaps(C.boundary, radius(N.theater->address, 100)) AND
       C.name = 'Andorra' AND 'Herbert the Worm' ∈ F.stars
```

Figure 25.3 Extended SQL to Find Herbert Films Playing near Andorra

The *theater* attribute of the Nowshowing table is a reference to an object in another table, which has attributes *name, address*, and *location*. This object referencing allows for the notation *N.theater->name* and *N.theater->address*, each of which refers to attributes of the `theater_t` object referenced in the Nowshowing row N. The *stars* attribute of the *films* table is a set of names of each film's stars. The *radius* method returns a circle centered at its first argument with radius equal to its second argument. The `overlaps` method tests for spatial overlap. Thus, Nowshowing and Films are joined by the equijoin clause, while Nowshowing and Countries are joined by the spatial overlap clause. The selections to 'Andorra' and films containing 'Herbert the Worm' complete the query.

These two object-relational queries are similar to SQL-92 queries but have some unusual features:

- **User-defined methods:** User-defined abstract types are manipulated via their methods, for example, *is_herbert* (Section 25.2).
- **Operators for structured types:** Along with the structured types available in the data model, ORDBMSs provide the natural methods for those types. For example, the **setof** types have the standard set methods $\in, \ni, \subset, \subseteq, =, \supseteq, \supset, \cup, \cap$, and $-$ (Section 25.3.1).
- **Operators for reference types:** Reference types are *dereferenced* via an arrow (->) notation (Section 25.4.2).

To summarize the points highlighted by our motivating example, traditional relational systems offer limited flexibility in the data types available. Data is stored in tables, and the type of each field value is limited to a simple atomic type (e.g., integer or string), with a small, fixed set of such types to choose from. This limited type system can be extended in three main ways: *user-defined abstract data types, structured types,* and *reference types.* Collectively, we refer to these new types as **complex types**. In the rest of this chapter we consider how a DBMS can be extended to provide support for defining new complex types and manipulating objects of these new types.

25.2 USER-DEFINED ABSTRACT DATA TYPES

Consider the Frames table of Figure 25.1. It has a column *image* of type `jpeg_image`, which stores a compressed image representing a single frame of a film. The `jpeg_image` type is not one of the DBMS's built-in types and was defined by a user for the Dinky application to store image data compressed using the JPEG standard. As another example, the Countries table defined in Line 7 of Figure 25.1 has a column *boundary* of type `polygon`, which contains representations of the shapes of countries' outlines on a world map.

Allowing users to define arbitrary new data types is a key feature of ORDBMSs. The DBMS allows users to store and retrieve objects of type `jpeg_image`, just like an object of any other type, such as `integer`. New atomic data types usually need to have type-specific operations defined by the user who creates them. For example, one might define operations on an image data type such as `compress`, `rotate`, `shrink`, and `crop`. The combination of an atomic data type and its associated methods is called an **abstract data type**, or **ADT**. Traditional SQL comes with built-in ADTs, such as integers (with the associated arithmetic methods), or strings (with the equality, comparison, and `LIKE` methods). Object-relational systems include these ADTs and also allow users to define their own ADTs.

The label 'abstract' is applied to these data types because the database system does not need to know how an ADT's data is stored nor how the ADT's methods work. It merely needs to know what methods are available and the input and output types for the methods. Hiding of ADT internals is called **encapsulation**.[1] Note that even in a relational system, atomic types such as integers have associated methods that are encapsulated into ADTs. In the case of integers, the standard methods for the ADT are the usual arithmetic operators and comparators. To evaluate the addition operator on integers, the database system need not understand the laws of addition—it merely needs to know how to invoke the addition operator's code and what type of data to expect in return.

In an object-relational system, the simplification due to encapsulation is critical because it hides any substantive distinctions between data types and allows an ORDBMS to be implemented without anticipating the types and methods that users might want to add. For example, adding integers and overlaying images can be treated uniformly by the system, with the only significant distinctions being that different code is invoked for the two operations and differently typed objects are expected to be returned from that code.

[1] Some ORDBMSs actually refer to ADTs as **opaque types** because they are encapsulated and hence one cannot see their details.

Packaged ORDBMS extensions: Developing a set of user-defined types and methods for a particular application—say image management—can involve a significant amount of work and domain-specific expertise. As a result, most ORDBMS vendors partner with third parties to sell prepackaged sets of ADTs for particular domains. Informix calls these extensions *DataBlades*, Oracle calls them *Data Cartridges*, IBM calls them *DB2 Extenders*, and so on. These packages include the ADT method code, DDL scripts to automate loading the ADTs into the system, and in some cases specialized access methods for the data type. Packaged ADT extensions are analogous to class libraries that are available for object-oriented programming languages: They provide a set of objects that together address a common task.

25.2.1 Defining Methods of an ADT

At a minimum, for each new atomic type a user must define methods that enable the DBMS to read in and to output objects of this type and to compute the amount of storage needed to hold the object. The user who creates a new atomic type must **register** the following methods with the DBMS:

- **Size**: Returns the number of bytes of storage required for items of the type or the special value *variable*, if items vary in size.
- **Import**: Creates new items of this type from textual inputs (e.g., `INSERT` statements).
- **Export**: Maps items of this type to a form suitable for printing, or for use in an application program (e.g., an ASCII string or a file handle).

In order to register a new method for an atomic type, users must write the code for the method and then inform the database system about the method. The code to be written depends on the languages supported by the DBMS, and possibly the operating system in question. For example, the ORDBMS may handle Java code in the Linux operating system. In this case the method code must be written in Java and compiled into a Java bytecode file stored in a Linux file system. Then an SQL-style method registration command is given to the ORDBMS so that it recognizes the new method:

```
CREATE FUNCTION is_sunrise(jpeg_image) RETURNS boolean
       AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
```

This statement defines the salient aspects of the method: the type of the associated ADT, the return type, and the location of the code. Once the method is registered,

the DBMS uses a Java virtual machine to execute the code[2]. Figure 25.4 presents a number of method registration commands for our Dinky database.

```
1. CREATE FUNCTION thumbnail(jpeg_image) RETURNS jpeg_image
       AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
2. CREATE FUNCTION is_sunrise(jpeg_image) RETURNS boolean
       AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
3. CREATE FUNCTION is_herbert(jpeg_image) RETURNS boolean
       AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
4. CREATE FUNCTION radius(polygon, float) RETURNS polygon
       AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
5. CREATE FUNCTION overlaps(polygon, polygon) RETURNS boolean
       AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
```

Figure 25.4 Method Registration Commands for the Dinky Database

Type definition statements for the user-defined atomic data types in the Dinky schema are given in Figure 25.5.

1. `CREATE ABSTRACT DATA TYPE jpeg_image`
   (*internallength* = `VARIABLE`, *input* = jpeg_in, *output* = jpeg_out);
2. `CREATE ABSTRACT DATA TYPE polygon`
   (*internallength* = `VARIABLE`, *input* = poly_in, *output* = poly_out);

Figure 25.5 Atomic Type Declaration Commands for Dinky Database

25.3 STRUCTURED TYPES

Atomic types and user-defined types can be combined to describe more complex structures using **type constructors**. For example, Line 6 of Figure 25.1 defines a column *stars* of type `setof(text)`; each entry in that column is a set of text strings, representing the stars in a film. The `setof` syntax is an example of a type constructor. Other common type constructors include:

- `ROW`(n_1 t_1, ..., n_n t_n): A type representing a row, or tuple, of n fields with fields $n_1, ..., n_n$ of types $t_1, ..., t_n$ respectively.
- `listof(base)`: A type representing a sequence of `base`-type items.
- `ARRAY(base)`: A type representing an array of `base`-type items.
- `setof(base)`: A type representing a *set* of `base`-type items. Sets cannot contain duplicate elements.

[2]In the case of non-portable compiled code – written, for example, in a language like C++ – the DBMS uses the operating system's dynamic linking facility to link the method code into the database system so that it can be invoked.

Structured data types in SQL: The `theater_t` type in Figure 25.1 illustrates the new `ROW` data type in SQL:1999; a value of `ROW` type can appear in a field of a tuple. In SQL:1999 the `ROW` type has a special role because every table is a collection of rows—every table is a set of rows or a multiset of rows. SQL:1999 also includes a data type called `ARRAY`, which allows a field value to be an array. The `ROW` and `ARRAY` type constructors can be freely interleaved and nested to build structured objects. The `listof`, `bagof`, and `setof` type constructors are not included in SQL:1999. IBM DB2, Informix UDS, and Oracle 8 support the `ROW` constructor.

- `bagof(base)`: A type representing a *bag* or *multiset* of `base`-type items.

To fully appreciate the power of type constructors, observe that they can be composed; for example, `ARRAY(ROW(`*age:* `integer`, *sal:* `integer`)). Types defined using type constructors are called **structured types**. Those using `listof`, `ARRAY`, `bagof`, or `setof` as the outermost type constructor are sometimes referred to as **collection types**, or **bulk data types**.

The introduction of structured types changes a fundamental characteristic of relational databases, which is that all fields contain atomic values. A relation that contains a structured type object is not in first normal form! We discuss this point further in Section 25.6.

25.3.1 Manipulating Data of Structured Types

The DBMS provides built-in methods for the types supported through type constructors. These methods are analogous to built-in operations such as addition and multiplication for atomic types such as integers. In this section we present the methods for various type constructors and illustrate how SQL queries can create and manipulate values with structured types.

Built-in Operators for Structured Types

We now consider built-in operators for each of the structured types that we presented in Section 25.3.

Rows: Given an item i whose type is `ROW(`n_1 $t_1, ..., n_n$ t_n`)`, the field extraction method allows us to access an individual field n_k using the traditional dot notation $i.n_k$. If row constructors are nested in a type definition, dots may be nested to access the fields of the nested row; for example $i.n_k.m_l$. If we have a collection of rows, the dot notation

gives us a collection as a result. For example, if i is a list of rows, $i.n_k$ gives us a list of items of type t_n; if i is a set of rows, $i.n_k$ gives us a set of items of type t_n.

This nested-dot notation is often called a **path expression** because it describes a path through the nested structure.

Sets and multisets: Set objects can be compared using the traditional set methods $\subset, \subseteq, =, \supseteq, \supset$. An item of type `setof(foo)` can be compared with an item of type `foo` using the $\in$ method, as illustrated in Figure 25.3, which contains the comparison *'Herbert the Worm'* $\in$ *F.stars*. Two set objects (having elements of the same type) can be combined to form a new object using the $\cup$, $\cap$, and $-$ operators.

Each of the methods for sets can be defined for multisets, taking the number of copies of elements into account. The $\cup$ operation simply adds up the number of copies of an element, the $\cap$ operation counts the lesser number of times a given element appears in the two input multisets, and $-$ subtracts the number of times a given element appears in the second multiset from the number of times it appears in the first multiset. For example, using multiset semantics $\cup(\{1,2,2,2\}, \{2,2,3\}) = \{1,2,2,2,2,2,3\}$; $\cap(\{1,2,2,2\}, \{2,2,3\}) = \{2,2\}$; and $-(\{1,2,2,2\}, \{2,2,3\}) = \{1,2\}$.

Lists: Traditional list operations include *head*, which returns the first element; *tail*, which returns the list obtained by removing the first element; *prepend*, which takes an element and inserts it as the first element in a list; and *append*, which appends one list to another.

Arrays: Array types support an 'array index' method to allow users to access array items at a particular offset. A postfix 'square bracket' syntax is usually used; for example, `foo_array[5]`.

Other: The operators listed above are just a sample. We also have the aggregate operators *count*, *sum*, *avg*, *max*, and *min*, which can (in principle) be applied to any object of a collection type. Operators for type conversions are also common. For example, we can provide operators to convert a multiset object to a set object by eliminating duplicates.

Examples of Queries Involving Nested Collections

We now present some examples to illustrate how relations that contain nested collections can be queried, using SQL syntax. Consider the Films relation. Each tuple describes a film, uniquely identified by *filmno*, and contains a set (of stars in the film) as a field value. Our first example illustrates how we can apply an aggregate operator

to such a nested set. It identifies films with more than two stars by counting the number of stars; the *count* operator is applied once per Films tuple.[3]

```
SELECT F.filmno
FROM   Films F
WHERE  count(F.stars) > 2
```

Our second query illustrates an operation called **unnesting**. Consider the instance of Films shown in Figure 25.6; we have omitted the *director* and *budget* fields (included in the Films schema in Figure 25.1) for simplicity. A flat version of the same information is shown in Figure 25.7; for each film and star in the film, we have a tuple in Films_flat.

filmno	*title*	*stars*
98	Casablanca	{Bogart, Bergman}
54	Earth Worms Are Juicy	{Herbert, Wanda}

Figure 25.6 A Nested Relation, Films

filmno	*title*	*star*
98	Casablanca	Bogart
98	Casablanca	Bergman
54	Earth Worms Are Juicy	Herbert
54	Earth Worms Are Juicy	Wanda

Figure 25.7 A Flat Version, Films_flat

The following query generates the instance of Films_flat from Films:

```
SELECT F.filmno, F.title, S AS star
FROM   Films F, F.stars AS S
```

The variable F is successively bound to tuples in Films, and for each value of F, the variable S is successively bound to the set in the *stars* field of F. Conversely, we may want to generate the instance of Films from Films_flat. We can generate the Films instance using a generalized form of SQL's `GROUP BY` construct, as the following query illustrates:

```
SELECT    F.filmno, F.title, set_gen(F.star)
FROM      Films_flat F
GROUP BY F.filmno, F.title
```

[3]SQL:1999 limits the use of aggregate operators on nested collections; to emphasize this restriction, we have used *count* rather than `COUNT`, which we reserve for legal uses of the operator in SQL.

Objects and oids: In SQL:1999 every tuple in a table can be given an oid by defining the table in terms of a structured type, as in the definition of the Theaters table in Line 4 of Figure 25.1. Contrast this with the definition of the Countries table in Line 7; Countries tuples do not have associated oids. SQL:1999 also assigns oids to large objects: this is the locator for the object.
There is a special type called `REF` whose values are the unique identifiers or oids. SQL:1999 requires that a given `REF` type must be associated with a specific structured type and that the table it refers to must be known at compilation time, i.e., the scope of each reference must be a table known at compilation time. For example, Line 5 of Figure 25.1 defines a column *theater* of type `ref(theater_t)`. Items in this column are references to objects of type `theater_t`, specifically the rows in the Theaters table, which is defined in Line 4. IBM DB2, Informix UDS, and Oracle 8 support `REF` types.

The operator *set_gen*, to be used with `GROUP BY`, requires some explanation. The `GROUP BY` clause partitions the Films_flat table by sorting on the *filmno* attribute; all tuples in a given partition have the same *filmno* (and therefore the same *title*). Consider the set of values in the *star* column of a given partition. This set cannot be returned in the result of an SQL-92 query, and we have to summarize it by applying an aggregate operator such as `COUNT`. Now that we allow relations to contain sets as field values, however, we would like to return the set of *star* values as a field value in a single answer tuple; the answer tuple also contains the *filmno* of the corresponding partition. The *set_gen* operator collects the set of *star* values in a partition and creates a set-valued object. This operation is called **nesting**. We can imagine similar generator functions for creating multisets, lists, and so on. However, such generators are not included in SQL:1999.

25.4 OBJECTS, OBJECT IDENTITY, AND REFERENCE TYPES

In object-database systems, data objects can be given an **object identifier (oid)**, which is some value that is unique in the database across time. The DBMS is responsible for generating oids and ensuring that an oid identifies an object uniquely over its entire lifetime. In some systems, all tuples stored in any table are objects and are automatically assigned unique oids; in other systems, a user can specify the tables for which the tuples are to be assigned oids. Often, there are also facilities for generating oids for larger structures (e.g., tables) as well as smaller structures (e.g., instances of data values such as a copy of the integer 5, or a JPEG image).

An object's oid can be used to refer (or 'point') to it from elsewhere in the data. Such a reference has a type (similar to the type of a pointer in a programming language), with a corresponding type constructor:

URLs and oids: It is instructive to note the differences between Internet URLs and the oids in object systems. First, oids uniquely identify a single object over all time, whereas the web resource pointed at by an URL can change over time. Second, oids are simply identifiers and carry no physical information about the objects they identify—this makes it possible to change the storage location of an object without modifying pointers to the object. In contrast, URLs include network addresses and often file-system names as well, meaning that if the resource identified by the URL has to move to another file or network address, then all links to that resource will either be incorrect or require a 'forwarding' mechanism. Third, oids are automatically generated by the DBMS for each object, whereas URLs are user-generated. Since users generate URLs, they often embed semantic information into the URL via machine, directory, or file names; this can become confusing if the object's properties change over time.
In the case of both URLs and oids, deletions can be troublesome: In an object database this can result in runtime errors during dereferencing; on the web this is the notorious '404 Page Not Found' error. The relational mechanisms for referential integrity are not available in either case.

`ref(base)`: a type representing a reference to an object of type `base`.

The `ref` type constructor can be interleaved with the type constructors for structured types; for example, `ROW(ref(ARRAY(integer)))`.

25.4.1 Notions of Equality

The distinction between reference types and reference-free structured types raises another issue: the definition of equality. Two objects having the same type are defined to be **deep equal** if and only if:

- The objects are of atomic type and have the same value, or
- The objects are of reference type, and the *deep equals* operator is true for the two referenced objects, or
- The objects are of structured type, and the *deep equals* operator is true for all the corresponding subparts of the two objects.

Two objects that have the same reference type are defined to be **shallow equal** if they both refer to the same object (i.e., both references use the same oid). The definition of shallow equality can be extended to objects of arbitrary type by taking the definition of deep equality and replacing *deep equals* by *shallow equals* in parts (2) and (3).

As an example, consider the complex objects `ROW(538, t89, 6-3-97,8-7-97)` and `ROW(538, t33, 6-3-97,8-7-97)`, whose type is the type of rows in the table Nowshowing (Line 5 of Figure 25.1). These two objects are not shallow equal because they differ in the second attribute value. Nonetheless, they might be deep equal, if, for instance, the oids **t89** and **t33** refer to objects of type `theater_t` that have the same value; for example, `tuple(54, 'Majestic', '115 King', '2556698')`.

While two deep equal objects may not be shallow equal, as the example illustrates, two shallow equal objects are always deep equal, of course. The default choice of deep versus shallow equality for reference types is different across systems, although typically we are given syntax to specify either semantics.

25.4.2 Dereferencing Reference Types

An item of reference type `ref(foo)` is not the same as the `foo` item to which it points. In order to access the referenced `foo` item, a built-in `deref()` method is provided along with the `ref` type constructor. For example, given a tuple from the Nowshowing table, one can access the *name* field of the referenced `theater_t` object with the syntax Nowshowing.`deref`*(theater).name*. Since references to tuple types are common, some systems provide a java-style arrow operator, which combines a postfix version of the dereference operator with a tuple-type dot operator. Using the arrow notation, the name of the referenced theater can be accessed with the equivalent syntax Nowshowing.*theater->name*, as in Figure 25.3.

At this point we have covered all the basic type extensions used in the Dinky schema in Figure 25.1. The reader is invited to revisit the schema and to examine the structure and content of each table and how the new features are used in the various sample queries.

25.5 INHERITANCE

We considered the concept of inheritance in the context of the ER model in Chapter 2 and discussed how ER diagrams with inheritance were translated into tables. In object-database systems, unlike relational systems, inheritance is supported directly and allows type definitions to be reused and refined very easily. It can be very helpful when modeling similar but slightly different classes of objects. In object-database systems, inheritance can be used in two ways: for reusing and refining types, and for creating hierarchies of collections of similar but not identical objects.

25.5.1 Defining Types with Inheritance

In the Dinky database, we model movie theaters with the type `theater_t`. Dinky also wants their database to represent a new marketing technique in the theater business: the *theater-cafe*, which serves pizza and other meals while screening movies. Theater-cafes require additional information to be represented in the database. In particular, a theater-cafe is just like a theater, but has an additional attribute representing the theater's menu. Inheritance allows us to capture this 'specialization' explicitly in the database design with the following DDL statement:

```
CREATE TYPE theatercafe_t UNDER theater_t (menu text);
```

This statement creates a new type, `theatercafe_t`, which has the same attributes and methods as `theater_t`, along with one additional attribute *menu* of type `text`. Methods defined on `theater_t` apply to objects of type `theatercafe_t`, but not vice versa. We say that `theatercafe_t` **inherits** the attributes and methods of `theater_t`.

Note that the inheritance mechanism is not merely a 'macro' to shorten `CREATE` statements. It creates an explicit relationship in the database between the **subtype** (`theatercafe_t`) and the **supertype** (`theater_t`): *An object of the subtype is also considered to be an object of the supertype.* This treatment means that any operations that apply to the supertype (methods as well as query operators such as projection or join) also apply to the subtype. This is generally expressed in the following principle:

> **The Substitution Principle**: Given a supertype A and a subtype B, it is always possible to substitute an object of type B into a legal expression written for objects of type A, without producing type errors.

This principle enables easy code reuse because queries and methods written for the supertype can be applied to the subtype without modification.

Note that inheritance can also be used for atomic types, in addition to row types. Given a supertype `image_t` with methods *title()*, *number_of_colors()*, and *display()*, we can define a subtype `thumbnail_image_t` for small images that inherits the methods of `image_t`.

25.5.2 Binding of Methods

In defining a subtype, it is sometimes useful to replace a method for the supertype with a new version that operates differently on the subtype. Consider the `image_t` type, and the subtype `jpeg_image_t` from the Dinky database. Unfortunately, the *display()* method for standard images does not work for JPEG images, which are specially compressed. Thus, in creating type `jpeg_image_t`, we write a special *display()* method

for JPEG images and register it with the database system using the CREATE FUNCTION command:

```
CREATE FUNCTION display(jpeg_image) RETURNS jpeg_image
        AS EXTERNAL NAME '/a/b/c/jpeg.class' LANGUAGE 'java';
```

Registering a new method with the same name as an old method is called **overloading** the method name.

Because of overloading, the system must understand which method is intended in a particular expression. For example, when the system needs to invoke the *display()* method on an object of type jpeg_image_t, it uses the specialized *display* method. When it needs to invoke *display* on an object of type image_t that is not otherwise subtyped, it invokes the standard *display* method. The process of deciding which method to invoke is called **binding** the method to the object. In certain situations, this binding can be done when an expression is parsed (**early binding**), but in other cases the most specific type of an object cannot be known until runtime, so the method cannot be bound until then (**late binding**). Late binding facilties add flexibility, but can make it harder for the user to reason about the methods that get invoked for a given query expression.

25.5.3 Collection Hierarchies, Type Extents, and Queries

Type inheritance was invented for object-oriented programming languages, and our discussion of inheritance up to this point differs little from the discussion one might find in a book on an object-oriented language such as C++ or Java.

However, because database systems provide query languages over tabular datasets, the mechanisms from programming languages are enhanced in object databases to deal with tables and queries as well. In particular, in object-relational systems we can define a table containing objects of a particular type, such as the Theaters table in the Dinky schema. Given a new subtype such as theater_cafe, we would like to create another table Theater_cafes to store the information about theater cafes. But when writing a query over the Theaters table, it is sometimes desirable to ask the same query over the Theater_cafes table; after all, if we project out the additional columns, an instance of the Theater_cafes table can be regarded as an instance of the Theaters table.

Rather than requiring the user to specify a separate query for each such table, we can inform the system that a new table of the subtype is to be treated as part of a table of the supertype, with respect to queries over the latter table. In our example, we can say:

```
CREATE TABLE Theater_cafes OF TYPE theater_cafe_t UNDER Theaters;
```

This statement tells the system that queries over the `theaters` table should actually be run over all tuples in both the `theaters` and `Theater_cafes` tables. In such cases, if the subtype definition involves method overloading, late-binding is used to ensure that the appropriate methods are called for each tuple.

In general, the `UNDER` clause can be used to generate an arbitrary tree of tables, called a **collection hierarchy**. Queries over a particular table T in the hierarchy are run over all tuples in T and its descendants. Sometimes, a user may want the query to run only on T, and not on the descendants; additional syntax, for example, the keyword `ONLY`, can be used in the query's `FROM` clause to achieve this effect.

Some systems automatically create special tables for each type, which contain references to every instance of the type that exists in the database. These tables are called **type extents** and allow queries over all objects of a given type, regardless of where the objects actually reside in the database. Type extents naturally form a collection hierarchy that parallels the type hierarchy.

25.6 DATABASE DESIGN FOR AN ORDBMS

The rich variety of data types in an ORDBMS offers a database designer many opportunities for a more natural or more efficient design. In this section we illustrate the differences between RDBMS and ORDBMS database design through several examples.

25.6.1 Structured Types and ADTs

Our first example involves several space probes, each of which continuously records a video. A single video stream is associated with each probe, and while this stream was collected over a certain time period, we assume that it is now a complete object associated with the probe. During the time period over which the video was collected, the probe's location was periodically recorded (such information can easily be 'piggy-backed' onto the header portion of a video stream conforming to the MPEG standard). Thus, the information associated with a probe has three parts: (1) a *probe id* that identifies a probe uniquely, (2) a *video stream*, and (3) a *location sequence* of ⟨*time, location*⟩ pairs. What kind of a database schema should we use to store this information?

An RDBMS Database Design

In an RDBMS, we must store each video stream as a BLOB and each location sequence as tuples in a table. A possible RDBMS database design is illustrated below:

Probes(*pid:* `integer`, *time:* `timestamp`, *lat:* `real`, *long:* `real`,

camera: `string`, *video:* `BLOB`)

There is a single table called Probes, and it has several rows for each probe. Each of these rows has the same *pid*, *camera*, and *video* values, but different *time*, *lat*, and *long* values. (We have used latitude and longitude to denote location.) The key for this table can be represented as a functional dependency: $PTLN \rightarrow CV$, where N stands for longitude. There is another dependency: $P \rightarrow CV$. This relation is therefore not in BCNF; indeed, it is not even in 3NF. We can decompose Probes to obtain a BCNF schema:

Probes_Loc(*pid:* `integer`, *time:* `timestamp`, *lat:* `real`, *long:* `real`)
Probes_Video(*pid:* `integer`, *camera:* `string`, *video:* `BLOB`)

This design is about the best we can achieve in an RDBMS. However, it suffers from several drawbacks.

First, representing videos as BLOBs means that we have to write application code in an external language to manipulate a video object in the database. Consider this query: "For probe 10, display the video recorded between 1:10 p.m. and 1:15 p.m. on May 10 1996." We have to retrieve the entire video object associated with probe 10, recorded over several hours, in order to display a segment recorded over 5 minutes.

Next, the fact that each probe has an associated sequence of location readings is obscured, and the sequence information associated with a probe is dispersed across several tuples. A third drawback is that we are forced to separate the video information from the sequence information for a probe. These limitations are exposed by queries that require us to consider all the information associated with each probe; for example, "For each probe, print the earliest time at which it recorded, and the camera type." This query now involves a join of Probes_Loc and Probes_Video on the *pid* field.

An ORDBMS Database Design

An ORDBMS supports a much better solution. First, we can store the video as an ADT object and write methods that capture any special manipulation that we wish to perform. Second, because we are allowed to store structured types such as lists, we can store the location sequence for a probe in a single tuple, along with the video information! This layout eliminates the need for joins in queries that involve both the sequence and video information. An ORDBMS design for our example consists of a single relation called Probes_AllInfo:

Probes_AllInfo(*pid:* `integer`, *locseq:* `location_seq`, *camera:* `string`,
video: `mpeg_stream`)

This definition involves two new types, `location_seq` and `mpeg_stream`. The `mpeg_stream` type is defined as an ADT, with a method *display()* that takes a start time and an end time and displays the portion of the video recorded during that interval. This method can be implemented efficiently by looking at the total recording duration and the total length of the video and interpolating to extract the segment recorded during the interval specified in the query.

Our first query is shown below in extended SQL syntax; using this *display* method: We now retrieve only the required segment of the video, rather than the entire video.

```
SELECT display(P.video, 1:10 p.m. May 10 1996, 1:15 p.m. May 10 1996)
FROM   Probes_AllInfo P
WHERE  P.pid = 10
```

Now consider the `location_seq` type. We could define it as a `list` type, containing a list of `ROW` type objects:

```
CREATE TYPE location_seq listof
       (row (time: timestamp, lat: real, long: real))
```

Consider the *locseq* field in a row for a given probe. This field contains a list of rows, each of which has three fields. If the ORDBMS implements collection types in their full generality, we should be able to extract the *time* column from this list to obtain a list of `timestamp` values, and to apply the `MIN` aggregate operator to this list to find the earliest time at which the given probe recorded. Such support for collection types would enable us to express our second query as shown below:

```
SELECT   P.pid, MIN(P.locseq.time)
FROM     Probes_AllInfo P
```

Current ORDBMSs are not as general and clean as this example query suggests. For instance, the system may not recognize that projecting the *time* column from a list of rows gives us a list of timestamp values; or the system may allow us to apply an aggregate operator only to a table and not to a nested list value.

Continuing with our example, we may want to do specialized operations on our location sequences that go beyond the standard aggregate operators. For instance, we may want to define a method that takes a time interval and computes the distance traveled by the probe during this interval. The code for this method must understand details of a probe's trajectory and geospatial coordinate systems. For these reasons, we might choose to define `location_seq` as an ADT.

Clearly, an (ideal) ORDBMS gives us many useful design options that are not available in an RDBMS.

25.6.2 Object Identity

We now discuss some of the consequences of using reference types or oids. The use of oids is especially significant when the size of the object is large, either because it is a structured data type or because it is a big object such as an image.

Although reference types and structured types seem similar, they are actually quite different. For example, consider a structured type `my_theater tuple(`*tno* `integer`, *name* `text`, *address* `text`, *phone* `text)` and the reference type `theater ref(theater_t)` of Figure 25.1. There are important differences in the way that database updates affect these two types:

- **Deletion:** Objects with references can be affected by the deletion of objects that they reference, while reference-free structured objects are not affected by deletion of other objects. For example, if the Theaters table were dropped from the database, an object of type `theater` might change value to *null*, because the `theater_t` object that it refers to has been deleted, while a similar object of type `my_theater` would not change value.
- **Update:** Objects of reference types will change value if the referenced object is updated. Objects of reference-free structured types change value only if updated directly.
- **Sharing versus copying:** An identified object can be referenced by multiple reference-type items, so that each update to the object is reflected in many places. To get a similar affect in reference-free types requires updating all 'copies' of an object.

There are also important storage distinctions between reference types and nonreference types, which might affect performance:

- **Storage overhead:** Storing copies of a large value in multiple structured type objects may use much more space than storing the value once and referring to it elsewhere through reference type objects. This additional storage requirement can affect both disk usage and buffer management (if many copies are accessed at once).
- **Clustering:** The subparts of a structured object are typically stored together on disk. Objects with references may point to other objects that are far away on the disk, and the disk arm may require significant movement to assemble the object and its references together. Structured objects can thus be more efficient than reference types if they are typically accessed in their entirety.

Many of these issues also arise in traditional programming languages such as C or Pascal, which distinguish between the notions of referring to objects *by value* and *by*

Oids and referential integrity: In SQL:1999, all the oids that appear in a column of a relation are required to reference the same target relation. This 'scoping' makes it possible to check oid references for 'referential integrity' just as foreign key references are checked. While current ORDBMS products supporting oids do not support such checks, it is likely that they will do so in future releases. This will make it much safer to use oids.

reference. In database design, the choice between using a structured type or a reference type will typically include consideration of the storage costs, clustering issues, and the effect of updates.

Object Identity versus Foreign Keys

Using an oid to refer to an object is similar to using a foreign key to refer to a tuple in another relation, but not quite the same: An oid can point to an object of `theater_t` that is stored *anywhere* in the database, even in a field, whereas a foreign key reference is constrained to point to an object in a particular referenced relation. This restriction makes it possible for the DBMS to provide much greater support for referential integrity than for arbitrary oid pointers. In general, if an object is deleted while there are still oid-pointers to it, the best the DBMS can do is to recognize the situation by maintaining a reference count. (Even this limited support becomes impossible if oids can be copied freely.) Thus, the responsibility for avoiding dangling references rests largely with the user if oids are used to refer to objects. This burdensome responsibility suggests that we should use oids with great caution and use foreign keys instead whenever possible.

25.6.3 Extending the ER Model

The ER model as we described it in Chapter 2 is not adequate for ORDBMS design. We have to use an extended ER model that supports structured attributes (i.e., sets, lists, arrays as attribute values), distinguishes whether entities have object ids, and allows us to model entities whose attributes include methods. We illustrate these comments using an extended ER diagram to describe the space probe data in Figure 25.8; our notational conventions are ad hoc, and only for illustrative purposes.

The definition of Probes in Figure 25.8 has two new aspects. First, it has a structured-type attribute `listof(row(`*time, lat, long*`))`; each value assigned to this attribute in a Probes entity is a list of tuples with three fields. Second, Probes has an attribute called videos that is an abstract data type object, which is indicated by a dark oval for this attribute with a dark line connecting it to Probes. Further, this attribute has an 'attribute' of its own, which is a method of the ADT.

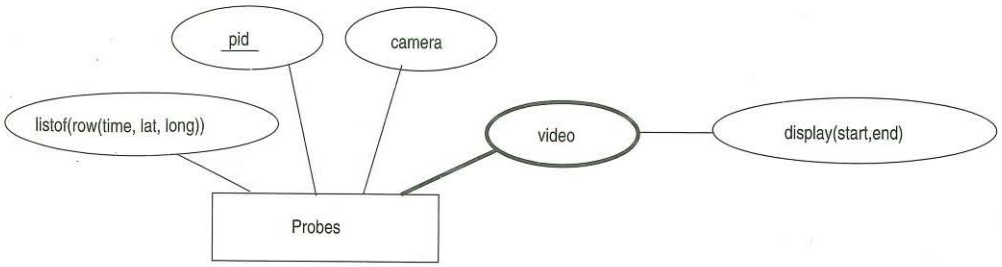


Figure 25.8 The Space Probe Entity Set

Alternatively, we could model each video as an entity by using an entity set called Videos. The association between Probes entities and Videos entities could then be captured by defining a relationship set that links them. Since each video is collected by precisely one probe, and every video is collected by some probe, this relationship can be maintained by simply storing a reference to a probe object with each Videos entity; this technique is essentially the second translation approach from ER diagrams to tables discussed in Section 2.4.1.

If we also make Videos a weak entity set in this alternative design, we can add a referential integrity constraint that causes a Videos entity to be deleted when the corresponding Probes entity is deleted. More generally, this alternative design illustrates a strong similarity between storing references to objects and foreign keys; the foreign key mechanism achieves the same effect as storing oids, but in a controlled manner. If oids are used, the user must ensure that there are no dangling references when an object is deleted, with very little support from the DBMS.

Finally, we note that a significant extension to the ER model is required to support the design of nested collections. For example, if a location sequence is modeled as an entity, and we want to define an attribute of Probes that contains a set of such entities, there is no way to do this without extending the ER model. We will not discuss this point further at the level of ER diagrams, but consider an example below that illustrates when to use a nested collection.

25.6.4 Using Nested Collections

Nested collections offer great modeling power, but also raise difficult design decisions. Consider the following way to model location sequences (other information about probes is omitted here to simplify the discussion):

Probes1(*pid:* `integer`, *locseq:* `location_seq`)

This is a good choice if the important queries in the workload require us to look at the location sequence for a particular probe, as in the query "For each probe, print the earliest time at which it recorded, and the camera type." On the other hand, consider a query that requires us to look at all location sequences: "Find the earliest time at which a recording exists for *lat=5, long=90*." This query can be answered more efficiently if the following schema is used:

Probes2(*pid:* `integer`, *time:* `timestamp`, *lat:* `real`, *long:* `real`)

The choice of schema must therefore be guided by the expected workload (as always!). As another example, consider the following schema:

Can_Teach1(*cid:* `integer`, *teachers:* `setof`(*ssn:* `string`), *sal:* `integer`)

If tuples in this table are to be interpreted as "Course *cid* can be taught by any of the teachers in the *teachers* field, at a cost *sal*" then we have the option of using the following schema instead:

Can_Teach2(*cid:* `integer`, *teacher_ssn:* `string`, *sal:* `integer`)

A choice between these two alternatives can be made based on how we expect to query this table. On the other hand, suppose that tuples in Can_Teach1 are to be interpreted as "Course *cid* can be taught by the team *teachers*, at a combined cost of *sal*." Can_Teach2 is no longer a viable alternative. If we wanted to flatten Can_Teach1, we would have to use a separate table to encode teams:

Can_Teach2(*cid:* `integer`, *team_id:* `oid`, *sal:* `integer`)
Teams(*tid:* `oid`, *ssn:* `string`)

As these examples illustrate, nested collections are appropriate in certain situations, but this feature can easily be misused; nested collections should therefore be used with care.

25.7 NEW CHALLENGES IN IMPLEMENTING AN ORDBMS

The enhanced functionality of ORDBMSs raises several implementation challenges. Some of these are well understood and solutions have been implemented in products; others are subjects of current research. In this section we examine a few of the key challenges that arise in implementing an efficient, fully functional ORDBMS. Many more issues are involved than those discussed here; the interested reader is encouraged to revisit the previous chapters in this book and consider whether the implementation techniques described there apply naturally to ORDBMSs or not.

25.7.1 Storage and Access Methods

Since object-relational databases store new types of data, ORDBMS implementors need to revisit some of the storage and indexing issues discussed in earlier chapters. In particular, the system must efficiently store ADT objects and structured objects and provide efficient indexed access to both.

Storing Large ADT and Structured Type Objects

Large ADT objects and structured objects complicate the layout of data on disk. This problem is well understood and has been solved in essentially all ORDBMSs and OODBMSs. We present some of the main issues here.

User-defined ADTs can be quite large. In particular, they can be bigger than a single disk page. Large ADTs, like BLOBs, require special storage, typically in a different location on disk from the tuples that contain them. Disk-based pointers are maintained from the tuples to the objects they contain.

Structured objects can also be large, but unlike ADT objects they often vary in size during the lifetime of a database. For example, consider the *stars* attribute of the *films* table in Figure 25.1. As the years pass, some of the 'bit actors' in an old movie may become famous.[4] When a bit actor becomes famous, Dinky might want to advertise his or her presence in the earlier films. This involves an insertion into the *stars* attribute of an individual tuple in *films*. Because these bulk attributes can grow arbitrarily, flexible disk layout mechanisms are required.

An additional complication arises with array types. Traditionally, array elements are stored sequentially on disk in a row-by-row fashion; for example

$$A_{11}, \ldots A_{1n}, A_{21}, \ldots, A_{2n}, \ldots A_{m1}, \ldots, A_{mn}$$

However, queries may often request subarrays that are not stored contiguously on disk (e.g., $A_{11}, A_{21}, \ldots, A_{m1}$). Such requests can result in a very high I/O cost for retrieving the subarray. In order to reduce the number of I/Os required in general, arrays are often broken into contiguous *chunks*, which are then stored in some order on disk. Although each chunk is some contiguous region of the array, chunks need not be row-by-row or column-by-column. For example, a chunk of size 4 might be $A_{11}, A_{12}, A_{21}, A_{22}$, which is a square region if we think of the array as being arranged row-by-row in two dimensions.

[4] A well-known example is Marilyn Monroe, who had a bit part in the Bette Davis classic *All About Eve*.

Indexing New Types

One important reason for users to place their data in a database is to allow for efficient access via indexes. Unfortunately, the standard RDBMS index structures support only equality conditions (B+ trees and hash indexes) and range conditions (B+ trees). An important issue for ORDBMSs is to provide efficient indexes for ADT methods and operators on structured objects.

Many specialized index structures have been proposed by researchers for particular applications such as cartography, genome research, multimedia repositories, Web search, and so on. An ORDBMS company cannot possibly implement every index that has been invented. Instead, the set of index structures in an ORDBMS should be user-extensible. Extensibility would allow an expert in cartography, for example, to not only register an ADT for points on a map (i.e., latitude/longitude pairs), but also implement an index structure that supports natural map queries (e.g., the R-tree, which matches conditions such as "Find me all theaters within 100 miles of Andorra"). (See Chapter 26 for more on R-trees and other spatial indexes.)

One way to make the set of index structures extensible is to publish an *access method interface* that lets users implement an index structure *outside* of the DBMS. The index and data can be stored in a file system, and the DBMS simply issues the *open*, *next*, and *close* iterator requests to the user's external index code. Such functionality makes it possible for a user to connect a DBMS to a Web search engine, for example. A main drawback of this approach is that data in an external index is not protected by the DBMS's support for concurrency and recovery. An alternative is for the ORDBMS to provide a generic 'template' index structure that is sufficiently general to encompass most index structures that users might invent. Because such a structure is implemented within the DBMS, it can support high concurrency and recovery. The *Generalized Search Tree* (GiST) is such a structure. It is a template index structure based on B+ trees, which allows most of the tree index structures invented so far to be implemented with only a few lines of user-defined ADT code.

25.7.2 Query Processing

ADTs and structured types call for new functionality in processing queries in ORDBMSs. They also change a number of assumptions that affect the efficiency of queries. In this section we look at two functionality issues (user-defined aggregates and security) and two efficiency issues (method caching and pointer swizzling).

User-Defined Aggregation Functions

Since users are allowed to define new methods for their ADTs, it is not unreasonable to expect them to want to define new aggregation functions for their ADTs as well. For example, the usual SQL aggregates—`COUNT`, `SUM`, `MIN`, `MAX`, `AVG`—are not particularly appropriate for the `image` type in the Dinky schema.

Most ORDBMSs allow users to register new aggregation functions with the system. To register an aggregation function, a user must implement three methods, which we will call *initialize*, *iterate*, and *terminate*. The *initialize* method initializes the internal state for the aggregation. The *iterate* method updates that state for every tuple seen, while the *terminate* method computes the aggregation result based on the final state and then cleans up. As an example, consider an aggregation function to compute the second-highest value in a field. The *initialize* call would allocate storage for the top two values, the *iterate* call would compare the current tuple's value with the top two and update the top two as necessary, and the *terminate* call would delete the storage for the top two values, returning a copy of the second-highest value.

Method Security

ADTs give users the power to add code to the DBMS; this power can be abused. A buggy or malicious ADT method can bring down the database server or even corrupt the database. The DBMS must have mechanisms to prevent buggy or malicious user code from causing problems. It may make sense to override these mechanisms for efficiency in production environments with vendor-supplied methods. However, it is important for the mechanisms to exist, if only to support debugging of ADT methods; otherwise method writers would have to write bug-free code before registering their methods with the DBMS—not a very forgiving programming environment!

One mechanism to prevent problems is to have the user methods be *interpreted* rather than *compiled*. The DBMS can check that the method is well behaved either by restricting the power of the interpreted language or by ensuring that each step taken by a method is safe before executing it. Typical interpreted languages for this purpose include Java and the procedural portions of SQL:1999.

An alternative mechanism is to allow user methods to be compiled from a general-purpose programming language such as C++, but to run those methods in a different address space than the DBMS. In this case the DBMS sends explicit interprocess communications (IPCs) to the user method, which sends IPCs back in return. This approach prevents bugs in the user methods (e.g., stray pointers) from corrupting the state of the DBMS or database and prevents malicious methods from reading or modifying the DBMS state or database as well. Note that the user writing the method need not know that the DBMS is running the method in a separate process: The user

code can be linked with a 'wrapper' that turns method invocations and return values into IPCs.

Method Caching

User-defined ADT methods can be very expensive to execute and can account for the bulk of the time spent in processing a query. During query processing it may make sense to cache the results of methods, in case they are invoked multiple times with the same argument. Within the scope of a single query, one can avoid calling a method twice on duplicate values in a column by either sorting the table on that column or using a hash-based scheme much like that used for aggregation (see Section 12.7). An alternative is to maintain a *cache* of method inputs and matching outputs as a table in the database. Then to find the value of a method on particular inputs, we essentially join the input tuples with the cache table. These two approaches can also be combined.

Pointer Swizzling

In some applications, objects are retrieved into memory and accessed frequently through their oids; dereferencing must be implemented very efficiently. Some systems maintain a table of oids of objects that are (currently) in memory. When an object O is brought into memory, they check each oid contained in O and replace oids of in-memory objects by in-memory pointers to those objects. This technique is called **pointer swizzling** and makes references to in-memory objects very fast. The downside is that when an object is paged out, in-memory references to it must somehow be invalidated and replaced with its oid.

25.7.3 Query Optimization

New indexes and query processing techniques widen the choices available to a query optimizer. In order to handle the new query processing functionality, an optimizer must know about the new functionality and use it appropriately. In this section we discuss two issues in exposing information to the optimizer (new indexes and ADT method estimation) and an issue in query planning that was ignored in relational systems (expensive selection optimization).

Registering Indexes with the Optimizer

As new index structures are added to a system—either via external interfaces or built-in template structures like GiSTs—the optimizer must be informed of their existence, and their costs of access. In particular, for a given index structure the optimizer must know (a) what `WHERE`-clause conditions are matched by that index, and (b) what the

Optimizer extensibility: As an example, consider the Oracle 8i optimizer, which is extensible and supports user defined 'domain' indexes and methods. The support includes user defined statistics and cost functions that the optimizer will use in tandem with system statistics. Suppose that there is a domain index for text on the *resume* column and a regular Oracle B-tree index on *hiringdate*. A query with a selection on both these fields can be evaluated by converting the rids from the two indexes into bitmaps, performing a bitmap AND, and converting the resulting bitmap to rids before accessing the table. Of course, the optimizer will also consider using the two indexes individually, as well as a full table scan.

cost of fetching a tuple is for that index. Given this information, the optimizer can use any index structure in constructing a query plan. Different ORDBMSs vary in the syntax for registering new index structures. Most systems require users to state a number representing the cost of access, but an alternative is for the DBMS to measure the structure as it is used and maintain running statistics on cost.

Reduction Factor and Cost Estimation for ADT Methods

In Section 14.2.1 we discussed how to estimate the reduction factor of various selection and join conditions including $=$, $<$, and so on. For user-defined conditions such as *is_herbert()*, the optimizer also needs to be able to estimate reduction factors. Estimating reduction factors for user-defined conditions is a difficult problem and is being actively studied. The currently popular approach is to leave it up to the user—a user who registers a method can also register an auxiliary function to estimate the method's reduction factor. If such a function is not registered, the optimizer uses an arbitrary value such as $\frac{1}{10}$.

ADT methods can be quite expensive and it is important for the optimizer to know just how much these methods cost to execute. Again, estimating method costs is open research. In current systems users who register a method are able to specify the method's cost as a number, typically in units of the cost of an I/O in the system. Such estimation is hard for users to do accurately. An attractive alternative is for the ORDBMS to run the method on objects of various sizes and attempt to estimate the method's cost automatically, but this approach has not been investigated in detail and is not implemented in commercial ORDBMSs.

Expensive selection optimization

In relational systems, selection is expected to be a zero-time operation. For example, it requires no I/Os and few CPU cycles to test if $emp.salary < 10$. However, conditions

such as *is_herbert(Frames.image)* can be quite expensive because they may fetch large objects off the disk and process them in memory in complicated ways.

ORDBMS optimizers must consider carefully how to order selection conditions. For example, consider a selection query that tests tuples in the Frames table with two conditions: $Frames.frameno < 100 \wedge is\_herbert(Frame.image)$. It is probably preferable to check the *frameno* condition before testing *is_herbert*. The first condition is quick and may often return false, saving the trouble of checking the second condition. In general, the best ordering among selections is a function of their costs and reduction factors. It can be shown that selections should be ordered by increasing *rank*, where rank = (reduction factor − 1)/cost. If a selection with very high rank appears in a multi-table query, it may even make sense to postpone the selection until after performing joins. Note that this approach is the opposite of the heuristic for pushing selections presented in Section 14.3! The details of optimally placing expensive selections among joins are somewhat complicated, adding to the complexity of optimization in ORDBMSs.

25.8 OODBMS

In the introduction of this chapter, we defined an OODBMS as a programming language with support for persistent objects. While this definition reflects the origins of OODBMSs accurately, and to a certain extent the implementation focus of OODBMSs, the fact that OODBMSs support *collection types* (see Section 25.3) makes it possible to provide a query language over collections. Indeed, a standard has been developed by the Object Database Management Group (ODMG) and is called **Object Query Language**, or **OQL**.

OQL is similar to SQL, with a `SELECT–FROM–WHERE`-style syntax (even `GROUP BY`, `HAVING`, and `ORDER BY` are supported) and many of the proposed SQL:1999 extensions. Notably, OQL supports structured types, including sets, bags, arrays, and lists. The OQL treatment of collections is more uniform than SQL:1999 in that it does not give special treatment to collections of rows; for example, OQL allows the aggregate operation `COUNT` to be applied to a list to compute the length of the list. OQL also supports reference types, path expressions, ADTs and inheritance, type extents, and SQL-style nested queries. There is also a standard Data Definition Language for OODBMSs (**Object Data Language**, or **ODL**) that is similar to the DDL subset of SQL, but supports the additional features found in OODBMSs, such as ADT definitions.

25.8.1 The ODMG Data Model and ODL

The ODMG data model is the basis for an OODBMS, just like the relational data model is the basis for an RDBMS. A database contains a collection of **objects**, which

Class = interface + implementation: Properly speaking, a class consists of an interface together with an implementation of the interface. An ODL interface definition is implemented in an OODBMS by translating it into declarations of the object-oriented language (e.g., C++, Smalltalk or Java) supported by the OODBMS. If we consider C++, for instance, there is a library of classes that implement the ODL constructs. There is also an **Object Manipulation Language (OML)** specific to the programming language (in our example, C++), which specifies how database objects are manipulated in the programming language. The goal is to seamlessly integrate the programming language and the database features.

are similar to entities in the ER model. Every object has a unique oid, and a database contains collections of objects with similar properties; such a collection is called a **class**.

The properties of a class are specified using ODL and are of three kinds: *attributes*, *relationships*, and *methods*. **Attributes** have an atomic type or a structured type. ODL supports the `set`, `bag`, `list`, `array`, and `struct` type constructors; these are just `setof`, `bagof`, `listof`, `ARRAY`, and `ROW` in the terminology of Section 25.3.

Relationships have a type that is either a reference to an object or a collection of such references. A relationship captures how an object is related to one or more objects of the same class or of a different class. A relationship in the ODMG model is really just a binary relationship in the sense of the ER model. A relationship has a corresponding **inverse relationship**; intuitively, it is the relationship 'in the other direction.' For example, if a movie is being shown at several theaters, and each theater shows several movies, we have two relationships that are inverses of each other: *shownAt* is associated with the class of movies and is the set of theaters at which the given movie is being shown, and *nowShowing* is associated with the class of theaters and is the set of movies being shown at that theater.

Methods are functions that can be applied to objects of the class. There is no analog to methods in the ER or relational models.

The keyword `interface` is used to define a class. For each interface, we can declare an **extent**, which is the name for the current set of objects of that class. The extent is analogous to the instance of a relation, and the interface is analogous to the schema. If the user does not anticipate the need to work with the set of objects of a given class—it is sufficient to manipulate individual objects—the extent declaration can be omitted.

The following ODL definitions of the Movie and Theater classes illustrate the above concepts. (While these classes bear some resemblance to the Dinky database schema, the reader should not look for an exact parallel, since we have modified the example to highlight ODL features.)

```
interface Movie
        (extent Movies key movieName)
        { attribute date start;
        attribute date end;
        attribute string moviename;
        relationship Set⟨Theater⟩ shownAt inverse Theater::nowShowing;
        }
```

The collection of database objects whose class is Movie is called Movies. No two objects in Movies have the same *movieName* value, as the key declaration indicates. Each movie is shown at a set of theaters and is shown during the specified period. (It would be more realistic to associate a different period with each theater, since a movie is typically played at different theaters over different periods. While we can define a class that captures this detail, we have chosen a simpler definition for our discussion.) A theater is an object of class Theater, which is defined below:

```
interface Theater
        (extent Theaters key theaterName)
        { attribute string theaterName;
        attribute string address;
        attribute integer ticketPrice;
        relationship Set⟨Movie⟩ nowShowing inverse Movie::shownAt;
        float numshowing() raises(errorCountingMovies);
        }
```

Each theater shows several movies and charges the same ticket price for every movie. Observe that the *shownAt* relationship of Movie and the *nowShowing* relationship of Theater are declared to be inverses of each other. Theater also has a method *numshowing()* that can be applied to a theater object to find the number of movies being shown at that theater.

ODL also allows us to specify inheritance hierarchies, as the following class definition illustrates:

```
interface SpecialShow extends Movie
        (extent SpecialShows)
        { attribute integer maximumAttendees;
        attribute string benefitCharity;
        }
```

An object of class SpecialShow is an object of class Movie, with some additional properties, as discussed in Section 25.5.

25.8.2 OQL

The ODMG query language OQL was deliberately designed to have syntax similar to SQL, in order to make it easy for users familiar with SQL to learn OQL. Let us begin with a query that finds pairs of movies and theaters such that the movie is shown at the theater and the theater is showing more than one movie:

```
SELECT mname: M.movieName, tname: T.theaterName
FROM   Movies M, M.shownAt T
WHERE  T.numshowing() > 1
```

The SELECT clause indicates how we can give names to fields in the result; the two result fields are called *mname* and *tname*. The part of this query that differs from SQL is the FROM clause. The variable M is bound in turn to each movie in the extent Movies. For a given movie M, we bind the variable T in turn to each theater in the collection *M.shownAt*. Thus, the use of the path expression *M.shownAt* allows us to easily express a nested query. The following query illustrates the grouping construct in OQL:

```
SELECT    T.ticketPrice,
          avgNum: AVG(SELECT P.T.numshowing() FROM partition P)
FROM      Theaters T
GROUP BY T.ticketPrice
```

For each ticket price, we create a group of theaters with that ticket price. This group of theaters is the partition for that ticket price and is referred to using the OQL keyword partition. In the SELECT clause, for each ticket price, we compute the average number of movies shown at theaters in the partition for that ticketPrice. OQL supports an interesting variation of the grouping operation that is missing in SQL:

```
SELECT    low, high,
          avgNum: AVG(SELECT P.T.numshowing() FROM partition P)
FROM      Theaters T
GROUP BY low: T.ticketPrice < 5, high: T.ticketPrice >= 5
```

The GROUP BY clause now creates just two partitions called *low* and *high*. Each theater object T is placed in one of these partitions based on its ticket price. In the SELECT clause, *low* and *high* are boolean variables, exactly one of which is true in any given output tuple; partition is instantiated to the corresponding partition of theater objects. In our example, we get two result tuples. One of them has *low* equal

to `true` and *avgNum* equal to the average number of movies shown at theaters with a low ticket price. The second tuple has *high* equal to `true` and *avgNum* equal to the average number of movies shown at theaters with a high ticket price.

The next query illustrates OQL support for queries that return collections other than set and multiset:

```
(SELECT   T.theaterName
FROM      Theaters T
ORDER BY  T.ticketPrice DESC) [0:4]
```

The `ORDER BY` clause makes the result a list of theater names ordered by ticket price. The elements of a list can be referred to by position, starting with position 0. Thus, the expression [0:4] extracts a list containing the names of the five theaters with the highest ticket prices.

OQL also supports `DISTINCT`, `HAVING`, explicit nesting of subqueries, view definitions, and other SQL features.

25.9 COMPARING RDBMS WITH OODBMS AND ORDBMS

Now that we have covered the main object-oriented DBMS extensions, it is time to consider the two main variants of object-databases, OODBMSs and ORDBMSs, and to compare them with RDBMSs. Although we have presented the concepts underlying object-databases, we still need to define the terms OODBMS and ORDBMS.

An **ORDBMS** is a relational DBMS with the extensions discussed in this chapter. (Not all ORDBMS systems support all the extensions in the general form that we have discussed them, but our concern in this section is the paradigm itself rather than specific systems.) An **OODBMS** is a programming language with a type system that supports the features discussed in this chapter and allows any data object to be **persistent**, that is, to survive across different program executions. Many current systems conform to neither definition entirely but are much closer to one or the other, and can be classified accordingly.

25.9.1 RDBMS versus ORDBMS

Comparing an RDBMS with an ORDBMS is straightforward. An RDBMS does not support the extensions discussed in this chapter. The resulting simplicity of the data model makes it easier to optimize queries for efficient execution, for example. A relational system is also easier to use because there are fewer features to master. On the other hand, it is less versatile than an ORDBMS.

25.9.2 OODBMS versus ORDBMS: Similarities

OODBMSs and ORDBMSs both support user-defined ADTs, structured types, object identity and reference types, and inheritance. Both support a query language for manipulating collection types. ORDBMSs support an extended form of SQL, and OODBMSs support ODL/OQL. The similarities are by no means accidental: ORDBMSs consciously try to add OODBMS features to an RDBMS, and OODBMSs in turn have developed query languages based on relational query languages. Both OODBMSs and ORDBMSs provide DBMS functionality such as concurrency control and recovery.

25.9.3 OODBMS versus ORDBMS: Differences

The fundamental difference is really a philosophy that is carried all the way through: OODBMSs try to add DBMS functionality to a programming language, whereas ORDBMSs try to add richer data types to a relational DBMS. Although the two kinds of object-databases are converging in terms of functionality, this difference in their underlying philosophies (and for most systems, their implementation approach) has important consequences in terms of the issues emphasized in the design of these DBMSs, and the efficiency with which various features are supported, as the following comparison indicates:

- OODBMSs aim to achieve seamless integration with a programming language such as C++, Java or Smalltalk. Such integration is not an important goal for an ORDBMS. SQL:1999, like SQL-92, allows us to embed SQL commands in a host language, but the interface is very evident to the SQL programer. (SQL:1999 also provides extended programming language constructs of its own, incidentally.)

- An OODBMS is aimed at applications where an object-centric viewpoint is appropriate; that is, typical user sessions consist of retrieving a few objects and working on them for long periods, with related objects (e.g., objects referenced by the original objects) fetched occasionally. Objects may be extremely large, and may have to be fetched in pieces; thus, attention must be paid to buffering parts of objects. It is expected that most applications will be able to cache the objects they require in memory, once the objects are retrieved from disk. Thus, considerable attention is paid to making references to in-memory objects efficient. Transactions are likely to be of very long duration and holding locks until the end of a transaction may lead to poor performance; thus, alternatives to Two Phase locking must be used.

  An ORDBMS is optimized for applications where large data collections are the focus, even though objects may have rich structure and be fairly large. It is expected that applications will retrieve data from disk extensively, and that optimizing disk accesses is still the main concern for efficient execution. Transactions are assumed

to be relatively short, and traditional RDBMS techniques are typically used for concurrency control and recovery.

- The query facilities of OQL are not supported efficiently in most OODBMSs, whereas the query facilities are the centerpiece of an ORDBMS. To some extent, this situation is the result of different concentrations of effort in the development of these systems. To a significant extent, it is also a consequence of the systems' being optimized for very different kinds of applications.

25.10 POINTS TO REVIEW

- *Object-oriented database systems (OODBMSs)* add DBMS functionality to a programming language and environment. *Object-relational database systems (ORDBMSs)* extend the functionality of relational database systems. The data type system is extended with *user-defined abstract data types (ADTs)*, *structured types*, and *inheritance*. New query features include *operators for structured types*, *operators for reference types*, and *user-defined methods*. **(Section 25.1)**

- An *abstract data type* is an atomic data type and its associated methods. Users can create new ADTs. For a new atomic type, we must *register* the methods *size*, *import*, and *export* with the DBMS. Object files containing new methods can also be registered. **(Section 25.2)**

- We can construct more complex data types from atomic types and user-defined types using *type constructors*. There are type constructors for creating *row types*, *lists*, *arrays*, *sets*, and *bags* and there are built-in operators for all these types. We can *unnest* a set-valued type by creating a tuple for each element in the set. The reverse operation is called *nesting*. **(Section 25.3)**

- Data objects can have an *object identifier (oid)*, which is a unique value that identifies the object. An oid has a type called a *reference type*. Since fields within objects can be of reference types, there are two notions of equality, *deep* and *shallow equality*. Fields that contain a reference type can be *dereferenced* to access the associated object. **(Section 25.4)**

- *Inheritance* allows us to create new types (called *subtypes*) that extend existing types (called *supertypes*). Any operations that apply to the supertype also apply to the subtype. We can *overload* methods by defining the same method for sub- and supertypes. The type of the object decides which method is called; this process is called *binding*. Analogous to types, we can create an inheritance hierarchy for tables called a *collection hierarchy*. **(Section 25.5)**

- The multitude of data types in an ORDBMS allows us to design a more natural and efficient database schema. But we have to be careful to take the differences between reference types and structured types and between reference types and

nonreference types into account. We should use oids with caution and use foreign keys instead whenever possible. We can extend the ER model to incorporate ADTs and methods, or we can model ADTs using existing constructs. Nested types provide great modeling power but the choice of schema should be guided by the expected workload. **(Section 25.6)**

- Implementing an ORDBMS brings new challenges. The system must store large ADTs and structured types that might be very large. Efficient and extensible index mechanisms must be provided. Examples of new functionality include *user-defined aggregation functions* (we can define new aggregation functions for our ADTs) and *method security* (the system has to prevent user-defined methods from compromising the security of the DBMS). Examples of new techniques to increase performance include *method caching* and *pointer swizzling*. The optimizer must know about the new functionality and use it appropriately. **(Section 25.7)**

- *Object Query Language (OQL)* is a query language for OODBMSs that provides constructs to query collection types. Its data definition language is called *Object Data Language (ODL)*. We discussed elements of the ODML data model and gave examples of OQL and ODL. **(Section 25.8)**

- An ORDBMS is a relational DBMS with the extensions discussed in this chapter. An OODBMS is a programming language with a type system that includes the features discussed in this chapter. ORDBMSs add OODBMS features to a RDBMS, but there are several differences to a full-fledged OODBMS. **(Section 25.9)**

EXERCISES

Exercise 25.1 Briefly answer the following questions.

1. What are the two kinds of new data types supported in object-database systems? Give an example of each, and discuss how the example situation would be handled if only an RDBMS was available.
2. What must a user do to define a new ADT?
3. Allowing users to define methods can lead to efficiency gains. Give an example.
4. What is late binding of methods? Give an example of inheritance that illustrates the need for dynamic binding.
5. What are collection hierarchies? Give an example that illustrates how collection hierarchies facilitate querying.
6. Discuss how a DBMS exploits encapsulation in implementing support for ADTs.
7. Give an example illustrating the nesting and unnesting operations.
8. Describe two objects that are deep equal but not shallow equal, or explain why this is not possible.

9. Describe two objects that are shallow equal but not deep equal, or explain why this is not possible.
10. Compare RDBMSs with ORDBMSs. Describe an application scenario for which you would choose an RDBMS, and explain why. Similarly, describe an application scenario for which you would choose an ORDBMS, and explain why.

Exercise 25.2 Consider the Dinky schema shown in Figure 25.1 and all related methods defined in the chapter. Write the following queries in extended SQL:

1. How many films were shown at theater *tno* = 5 between January 1 and February 1 of 1997?
2. What is the lowest budget for a film with at least two stars?
3. Consider theaters at which a film directed by Steven Spielberg started showing on January 1, 1997. For each such theater, print the names of all countries within a 100-mile radius. (You can use the *overlap* and *radius* methods illustrated in Figure 25.2.)

Exercise 25.3 In a company database, you need to store information about employees, departments, and children of employees. For each employee, identified by *ssn*, you must record *years* (the number of years that the employee has worked for the company), *phone*, and *photo* information. There are two subclasses of employees: contract and regular. Salary is computed by invoking a method that takes *years* as a parameter; this method has a different implementation for each subclass. Further, for each regular employee, you must record the name and age of every child. The most common queries involving children are similar to "Find the average age of Bob's children" and "Print the names of all of Bob's children."

A photo is a large image object and can be stored in one of several image formats (e.g., gif, jpeg). You want to define a *display* method for image objects; display must be defined differently for each image format. For each department, identified by *dno*, you must record *dname*, *budget*, and *workers* information. *Workers* is the set of employees who work in a given department. Typical queries involving workers include, "Find the average salary of all workers (across all departments)."

1. Using extended SQL, design an ORDBMS schema for the company database. Show all type definitions, including method definitions.
2. If you have to store this information in an RDBMS, what is the best possible design?
3. Compare the ORDBMS and RDBMS designs.
4. If you are told that a common request is to display the images of all employees in a given department, how would you use this information for physical database design?
5. If you are told that an employee's image must be displayed whenever any information about the employee is retrieved, would this affect your schema design?
6. If you are told that a common query is to find all employees who look similar to a given image, and given code that lets you create an index over all images to support retrieval of similar images, what would you do to utilize this code in an ORDBMS?

Exercise 25.4 ORDBMSs need to support efficient access over collection hierarchies. Consider the collection hierarchy of Theaters and Theater_cafes presented in the Dinky example. In your role as a DBMS implementor (not a DBA), you must evaluate three storage alternatives for these tuples:

- All tuples for all kinds of theaters are stored together on disk in an arbitrary order.
- All tuples for all kinds of theaters are stored together on disk, with the tuples that are from Theater_cafes stored directly after the last of the non-cafe tuples.
- Tuples from Theater_cafes are stored separately from the rest of the (non-cafe) theater tuples.

1. For each storage option, describe a mechanism for distinguishing plain theater tuples from Theater_cafe tuples.
2. For each storage option, describe how to handle the insertion of a new non-cafe tuple.
3. Which storage option is most efficient for queries over all theaters? Over just Theater_cafes? In terms of the number of I/Os, how much more efficient is the best technique for each type of query compared to the other two techniques?

Exercise 25.5 Different ORDBMSs use different techniques for building indexes to evaluate queries over collection hierarchies. For our Dinky example, to index theaters by name there are two common options:

- Build one B+ tree index over Theaters.*name* and another B+ tree index over Theater_cafes.*name*.
- Build one B+ tree index over the union of Theaters.*name* and Theater_cafes.*name*.

1. Describe how to efficiently evaluate the following query using each indexing option (this query is over all kinds of theater tuples):

   ```
   SELECT * FROM Theaters T WHERE T.name = 'Majestic'
   ```

   Give an estimate of the number of I/Os required in the two different scenarios, assuming there are 1,000,000 standard theaters and 1,000 theater-cafes. Which option is more efficient?
2. Perform the same analysis over the following query:

   ```
   SELECT * FROM Theater_cafes T WHERE T.name = 'Majestic'
   ```

3. For clustered indexes, does the choice of indexing technique interact with the choice of storage options? For unclustered indexes?

Exercise 25.6 Consider the following query:

```
SELECT thumbnail(I.image)
FROM   Images I
```

Given that the *I.image* column may contain duplicate values, describe how to use hashing to avoid computing the *thumbnail* function more than once per distinct value in processing this query.

Exercise 25.7 You are given a two-dimensional, $n \times n$ array of objects. Assume that you can fit 100 objects on a disk page. Describe a way to lay out (chunk) the array onto pages so that retrievals of square $m \times m$ subregions of the array are efficient. (Different queries will request subregions of different sizes, i.e., different m values, and your arrangement of the array onto pages should provide good performance, on average, for all such queries.)

Exercise 25.8 An ORDBMS optimizer is given a single-table query with n expensive selection conditions, $\sigma_n(...(\sigma_1(T)))$. For each condition σ_i, the optimizer can estimate the cost c_i of evaluating the condition on a tuple and the reduction factor of the condition r_i. Assume that there are t tuples in T.

1. How many tuples appear in the output of this query?
2. Assuming that the query is evaluated as shown (without reordering selections), what is the total cost of the query? Be sure to include the cost of scanning the table and applying the selections.
3. In Section 25.7.2 it was asserted that the optimizer should reorder selections so that they are applied to the table in order of increasing rank, where $\text{rank}_i = (r_i - 1)/c_i$. Prove that this assertion is optimal. That is, show that no other ordering could result in a query of lower cost. (Hint: It may be easiest to consider the special case where $n = 2$ first and generalize from there.)

BIBLIOGRAPHIC NOTES

A number of the object-oriented features described here are based in part on fairly old ideas in the programming languages community. [35] provides a good overview of these ideas in a database context. Stonebraker's book [630] describes the vision of ORDBMSs embodied by his company's early product, Illustra (now a product of Informix). Current commercial DBMSs with object-relational support include Informix Universal Server, IBM DB/2 CS V2, and UniSQL. An new version of Oracle is scheduled to include ORDBMS features as well.

Many of the ideas in current object-relational systems came out of a few prototypes built in the 1980s, especially POSTGRES [633], Starburst [296], and O2 [183].

The idea of an object-oriented database was first articulated in [165], which described the GemStone prototype system. Other prototypes include DASDBS [571], EXODUS [110], IRIS [235], ObjectStore [400], ODE, [14] ORION [370], SHORE [109], and THOR [417]. O2 is actually an early example of a system that was beginning to merge the themes of ORDBMSs and OODBMSs—it could fit in this list as well. [34] lists a collection of features that are generally considered to belong in an OODBMS. Current commercially available OODBMSs include GemStone, Itasca, O2, Objectivity, ObjectStore, Ontos, Poet, and Versant. [369] compares OODBMSs and RDBMSs.

Database support for ADTs was first explored in the INGRES and POSTGRES projects at U.C. Berkeley. The basic ideas are described in [627], including mechanisms for query processing and optimization with ADTs as well as extensible indexing. Support for ADTs was also investigated in the Darmstadt database system, [415]. Using the POSTGRES index extensibility correctly required intimate knowledge of DBMS-internal transaction mechanisms. Generalized search trees were proposed to solve this problem; they are described in [317], with concurrency and ARIES-based recovery details presented in [386]. [585] proposes that users must be allowed to define operators over ADT objects and properties of these operators that can be utilized for query optimization, rather than just a collection of methods.

Array chunking is described in [568]. Techniques for method caching and optimizing queries with expensive methods are presented in [315, 140]. Client-side data caching in a client-server OODBMS is studied in [242]. Clustering of objects on disk is studied in [650]. Work on nested relations was an early precursor of recent research on complex objects in OODBMSs and ORDBMSs. One of the first nested relation proposals is [439]. MVDs play an important role in reasoning about reduncancy in nested relations; see, for example, [504]. Storage structures for nested relations were studied in [181].

Formal models and query languages for object-oriented databases have been widely studied; papers include [4, 46, 62, 105, 331, 332, 366, 503, 635]. [365] proposes SQL extensions for querying object-oriented databases. An early and elegant extension of SQL with path expressions and inheritance was developed in GEM [692]. There has been much interest in combining deductive and object-oriented features. Papers in this area include [37, 247, 430, 485, 617, 694]. See [3] for a thorough textbook discussion of formal aspects of object-orientation and query languages.

[371, 373, 634, 696] include papers on DBMSs that would now be termed object-relational and/or object-oriented. [695] contains a detailed overview of schema and database evolution in object-oriented database systems. Drafts of the SQL3 standard are available electronically at URL `ftp://jerry.ece.umassd.edu/isowg3/`. The SQL:1999 standard is discussed in [200]. The incorporation of several SQL:1999 features into IBM DB2 is described in [108]. OQL is described in [120]. It is based to a large extent on the O2 query language, which is described, together with other aspects of O2, in the collection of papers [45].



Ramakrishnan
Gehrke

Database Management Sys

Second Edition

T1-AQK-423

777

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